Flatten and Conquer

A Framework for Efficient Analysis of String Constraints

Parosh Aziz Abdulla
Uppsala University, Sweden
parosh@it.uu.se

Mohamed Faouzi Atig
Uppsala University, Sweden
mohamed_faouzi.atig@it.uu.se

Yu-Fang Chen
Academia Sinica, Taiwan
yfc@iis.sinica.edu.tw

Bui Phi Diep
Uppsala University, Sweden
bui.phi-diep@it.uu.se

Lukáš Holík
Brno University of Technology,
Czech Republic
holik@fit.vutbr.cz

Ahmed Rezine
Linköping University, Sweden
ahmed.rezine@liu.se

Philipp Rümmer
Uppsala University, Sweden
philipp.ruemmer@it.uu.se

Abstract

We describe a uniform and efficient framework for checking the satisfiability of a large class of string constraints. The framework is based on the observation that both satisfiability and unsatisfiability of common constraints can be demonstrated through witnesses with simple patterns. These patterns are captured using *flat* automata each of which consists of a sequence of simple loops. We build a Counter-Example Guided Abstraction Refinement (CEGAR) framework which contains both an under- and an over-approximation module. The flow of information between the modules allows to increase the precision in an automatic manner. We have implemented the framework as a tool and performed extensive experimentation that demonstrates both the generality and efficiency of our method.

CCS Concepts • **Security and privacy** → **Logic and verification**; • **Software and its engineering** → *Formal methods*

Keywords String Equation; Formal Verification; Automata Theory

PLDI'17, June 18–23, 2017, Barcelona, Spain
© 2017 ACM. 978-1-4503-4988-8/17/06...$15.00
http://dx.doi.org/10.1145/3062341.3062384

1. Introduction

Background. There has been a substantial amount of research in recent years on the development of solvers for *string constraints* [3, 25, 38, 43, 50]. This has been motivated by numerous application areas such as security, web programming, and model checking. For instance, cross-site scripting (XSS), one of the most common types of web vulnerabilities, may be used by attackers to bypass access controls and is typically caused by improper handling of strings by web applications [27]. Verification techniques such as regular model checking [1], use string constraints as symbolic encodings of infinite sets of program states.

A major difficulty in the analysis of string manipulating programs is that any reasonably comprehensive theory over strings is either undecidable or difficult to the degree that the decidability problem has been open for many years [8, 17, 18, 31]. Therefore, existing string solver tools handle only fragments of the theory of strings and regular languages, sometimes with strong restrictions on the expressiveness and the input language. Another source of difficulty is the diversity of the application areas which means that string constraints come in very different forms.

It is not trivial to combine solutions for different types of constraints in a single framework. In fact, many classes of constraints, such as membership in context-free grammars and transducers are not supported by current tools. This represents an important limitation for several applications. Indeed, to mention a few examples, the ability to reason about context-free grammars and transducers is crucial for precisely detecting SQL and command injections in web ap-

plications [41, 47], for comparing grammars or reasoning about the ambiguities or correctness of parsers [29], or for enabling deeper symbolic testing [22]. For instance, SQL injections occur when valid SQL queries (i.e. words belonging to a specific context-free grammar) are built from subwords with a meaning that is different from the one intended by the programmer (which could also be expressed in terms of a context-free membership constraint [41]). In addition, precisely tracking all possible queries requires the ability to capture the effect of string-manipulating operations using transducers, word equations, and length constraints.

Framework. We propose a novel technique, called *flattening*, to solve the satisfiability problem for string constraints. A flat automaton is defined by an *abstraction parameter* consisting of a pair $\alpha = \langle \mathbb{p}, \mathbb{q} \rangle$ of natural numbers. A run of a flat automaton iterates a sequence of $\mathbb{q}$ loops each corresponding to a fixed word of length at most $\mathbb{p}$ (see Fig. 2). Flattening of a constraint means that we perform an *under-approximation* in which we restrict the search for solutions to only those strings that are generated by a flat automaton.

We build our framework using a classical concept from language theory, namely that of *Parikh images* [33]. The Parikh image of a word over a given alphabet counts the number of occurrences of each symbol in the word without regard to their order. The Parikh image of a language is the set of Parikh images of the words in the language. We say that a language is Parikh-definable, if its Parikh image is computable as a quantifier-free Presburger formula (linear arithmetic). If a language is Parikh-definable, then we can use an SMT-solver to check its emptiness. More precisely, we first compute its Parikh image as a quantifier-free Presburger formula. Since SMT-solvers can check the satisfiability of such formulas, we can feed the generated formula to the SMT-solver. The language is empty if and only if the SMT-solver concludes that the formula is unsatisfiable.

The framework is applicable to any class of constraints satisfying a *sufficient* condition which states that the flattening of any constraint is Parikh-definable. We show that this condition is satisfied by a wide class of constraints. For instance, for a constraint ϕ that requires membership in a context-free grammar $\mathcal{G}$, we show that we can derive a new grammar $\mathcal{G}'$ that captures the flattening of ϕ (i.e., the set of strings that are accepted by $\mathcal{G}$ and by the flat automaton). Then our sufficient condition follows since context-free languages are Parikh-definable in general, and hence in particular (the language of) $\mathcal{G}'$ is Parikh-definable. Furthermore, we show that the flattening of a word equation can be captured using a finite-state automaton. This implies our sufficient condition by Parikh-definability of regular languages. In fact, using a similar pattern, we can cover all kinds of string constraints known to us from applications, including word equations, length constraints, membership in context-free grammars, and transducer relations. We show that the flattening operation can be performed in polynomial time. It is well-known that computing the Parikh image of constraints in the above form can be performed in polynomial time. Thus, our scheme translates in a uniform way and in polynomial time the satisfiability of a flat constraint to the satisfiability of a quantifier-free Presburger formula. This allows the leveraging of available powerful SMT-solvers for linear arithmetic such as Z3 [11], CVC4 [6], Princess [7], MathSat [9], or Yices [12].

Flat automata enjoy two properties that make them attractive for the analysis of string constraints. First, the simplicity of the structure of flat automata allows efficient computation of their products with string constraints. Second, as we demonstrate through our experiments, although solutions to string constraints may have large sizes, they usually follow simple patterns that can easily be captured by flat automata that are small in size, thus making the analysis extremely efficient compared to existing tools.

Based on flat automata, we have developed a Counter-Example Guided Refinement (CEGAR) framework in which two procedures are run in an alternating manner: one that considers an under-approximation of the input constraints, based on satisfiability checking; and one that considers an over-approximation, based on unsatisfiability checking. The approximations are refined on demand by letting information flow between the two modules. More precisely, if the under-approximation fails to find a solution for a given set of abstraction parameters, then this information is used for excluding an infinite set of solutions when performing the next over-approximation. Furthermore, if the over-approximation produces a counter-example then it can be used to adjust the abstraction parameters so that the counter-example is not regenerated during subsequent iterations of the procedure.

We have implemented our framework in an open source solver, called TRAU[1], using Z3 [11] as an SMT solver. We are not aware of other solvers that can handle the same set of string constraints without restricting the lengths of the solutions. Therefore, we have evaluated TRAU using two separate sets of benchmarks. First, we compare TRAU against existing state-of-the-art solvers for string equations with length and regular constraints using the Kaluza benchmarks [38]. Then, we use a set of string constraints with CFG queries in order to verify the absence of SQL injections. The experiments demonstrate both the generality and efficiency of our method.

Summary of Contributions.

- A fundamentally new method for checking satisfiability of string constraints based on the concept of flattening. The method is general and allows the handling of all classes of constraints known to us from applications.
- An algorithm that translates the satisfiability of flat constraints to the satisfiability of quantifier-free Presburger formulas, thus allowing the use of powerful SMT solvers.

[1] TRAU, pronounced /chow/, is a *buffalo* - a mascot in Vietnamese culture.

- A CEGAR framework that allows the flow of information between an under- and over-approximation module, leading to more and more precise approximations.
- Implementation of an open source tool with experimental results that demonstrate the efficiency and generality of our approach on both existing and original benchmarks.

2. Related Work

Already in 1946, Quine [36] showed that the first order theory of string equations is undecidable. An important line of work has been to identify subclasses for which decidability can be achieved. The pioneering work by Makanin [30] proposed a decision procedure for quantifier-free word equations, i.e., Boolean combinations of equalities and disequalities, where the variables may denote words of arbitrary lengths. The decidability and complexity of different subclasses have been considered by several works, e.g. [17, 18, 31, 34, 35, 37, 40].

Generalizations of the work of Makanin by adding new types of constraints have been difficult to achieve. For instance, the satisfiability of word equations combined with length constraints of the form $|x| = |y|$ is open [8]. The problem of checking satisfiability for the class of constraints we consider in this paper is undecidable due to having context-free grammars and transducers.

Over the last years, several SMT solvers for strings and related logics have been introduced, applying a variety of calculi and algorithms. A number of tools handle string constraints, including context-free grammars, by means of *length-based under-approximation* and translation to bit-vectors [24, 38, 39], assuming a fixed upper bound on the length of the possible solutions. Our under-approximation of string constraints using flat automata is more powerful since we can find solutions of unbounded length; in addition, in our work also over-approximations are used to show unsatisfiability.

More recently, also *DPLL(T)-based* string solvers lift the restriction to strings of bounded length; this generation of solvers includes Z3-str2 [50], CVC4 [25], S3 [43], and Norn [3]. DPLL(T)-based solvers handle a variety of string constraints, including word equations, regular expression membership, length constraints, and (more rarely) regular/rational relations; the solvers are not complete for the full combination of those constraints though, and often only decide a (more or less well-defined) fragment of the individual constraints. Equality constraints are normally handled by means of splitting into simpler sub-cases, in combination with powerful techniques for Boolean reasoning to curb the resulting exponential search space. A recent paper [27] also proposes a splitting-based method to solve relational constraints defined by transducers. In comparison, our framework handles a larger set of constraints, including context-free grammars and transducers, and proposes a novel approximation scheme that avoids splitting of equations altogether. Splitting of equations can cause an explosion in the number of cases to be investigated by solvers.

A further direction is *automata-based* solvers for analyzing string-manipulated programs. Stranger [48] soundly over-approximates string constraints using regular languages, and outperforms DPLL(T)-based solvers when checking single execution traces, according to some evaluations [23]. It has recently also been observed [46] that automata-based algorithms can be combined with model checking algorithms, in particular IC3/PDR, for more efficient checking of the emptiness for automata. However, many kinds of constraints, including length constraints, word equations, and context-free grammars, cannot be handled by automata-based solvers in a complete manner. Our framework uses flat automata to define both over- and under-approximations of constraints, but not to represent string constraints in their entirety. Thus we remove some of the main limitations of previous automata-based approaches: a larger range of constraints can be handled, and satisfying assignments can be computed.

Flat automata (or equivalently bounded languages [20, 21]) have been also used in the context of verification of concurrent recursive programs (e.g., [5, 13, 15, 16, 19, 28]). In particular the work [28] uses a similar CEGAR approach for the verification of safety properties for concurrent recursive programs. However, the application of the CEGAR approach to the case of string constraints raises several new challenges since it requires (1) new methods for checking satisfiability of string constraints based on the concept of flattening and (2) new over-approximation techniques. To the best of our knowledge, such CEGAR frameworks have not been applied for string solving.

3. Overview

We give an overview of the framework and illustrate the main ingredients using a simple example.

Framework. Our procedure for solving string constraints is depicted in Fig. 1. The procedure inputs a set ψ of string constraints. If it terminates then it either returns the value $\perp$ which means that ψ is unsatisfiable, or it returns a solution v to ψ. In general, termination is not guaranteed. This is to be expected since the problem of solving string constraints, in the general class considered in this paper, is undecidable.

The procedure consists of a sequence of under- and over-approximation phases, one followed by the other. We maintain a set *Waiting* of abstraction parameters, to be considered by the under-approximation module in the coming iterations. Each iteration of the under-approximation module selects and removes one such a parameter α from the set. The parameter α is moved to the set *Covered* that contains all the abstraction parameters that have already been considered by the under-approximation. The procedure flattens the input set ψ wrt. α, and computes its Parikh image as a quantifier-free Presburger formula ρ which is given to the

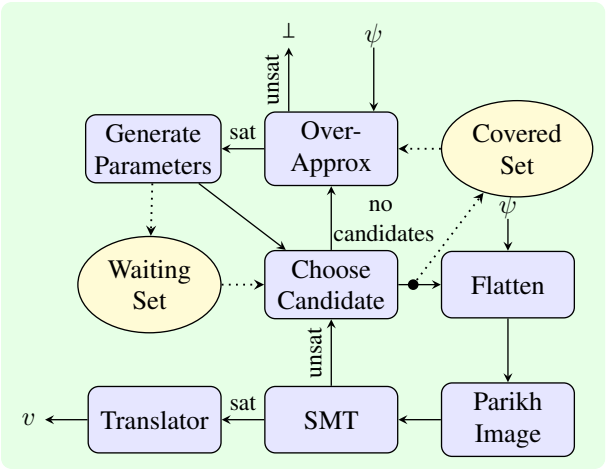

Figure 1. Overview of Framework

SMT solver. If ρ is satisfiable, then the SMT solver will output a satisfying assignment that is translated to a satisfying assignment v of ψ. The assignment v is output to the user and the procedure terminates. On the other hand, if the SMT solver concludes that there is no satisfying assignment then the under-approximation module fetches the next parameter from the set *Waiting* and repeats the loop. This continues until either a satisfying assignment is found or the set *Waiting* becomes empty. In the latter case, we have run out of parameters. This ends the current under-approximation phase, and triggers an over-approximation phase.

The over-approximation procedure uses the set *Covered* to prune the search space of solutions. More precisely, at this stage, we know that the under-approximation has checked satisfiability for all the current elements of *Covered*. Therefore, we know that ψ is not satisfiable for any one of them. Consequently, the over-approximation needs only to search for solutions outside the languages of the corresponding flat automata. If the over-approximation does not find a solution, then we know that ψ is unsatisfiable, and the procedure can terminate. However, if the over-approximation finds a solution v, then we need to check whether v is a spurious or genuine solution of ψ. This can be done by simply running the under-approximation on v. However, in order to increase efficiency, we use v to accelerate the under-approximation. To that end, we generate the minimal elements of the set of all abstraction parameters whose corresponding automata accept v, and put them in the set *Waiting*. Our experiments indicate that these minimal elements have often small values even for long strings. Now, the over-approximation phase terminates and the next under-approximation phase starts.

Notice that parameters that have been added to the set *Waiting* ensure the potential solution v will be considered by the under-approximation. In fact, if v is a genuine solution then this will be detected by the under-approximation in the next phase. If the under-approximation fails to find a solution even during the next phase, then since we move all the new parameters to the set *Covered*, v will not be re-generated in the subsequent phases by the over-approximation.

Observe that the flow of information between the two modules is carried out using the sets *Waiting* and *Covered*. The parameters considered by the under-approximation are used to prune an infinite set from the state space searched by the over-approximation. Also, spurious counter-examples provided by the over-approximation generate new sets of parameters on which the under-approximation can be performed.

The framework is not dependent on the particular over-approximation scheme used. In fact, any algorithm which returns a potential solution to ψ is sufficient for our purposes. In this paper, we considered a simple over-approximation scheme which consists in: (1) Replacing a membership constraint in a context-free grammar $\mathcal{G}$ by a membership constraint in a regular language . The regular language may be the upward closure of the language of $\mathcal{G}$ [4, 45], the downward closure [10, 44], or some other over-approximation, e.g., the one in [32]. (2) Replacing a transducer constraint by a conjunction of membership constraints in regular languages where each regular language captures the projection of the transducer language on one of its tapes, and (3) Ensuring that there are no cyclic dependencies among variables that appear in the set of (dis-)equality constraints [2]. This is done by replacing any occurrence of a variable x by a fresh copy that satisfies the same membership and length constraints as x. The resulting set of string constraints ψ falls in the decidable fragment of the theory of strings with regular and length constraints on which a similar technique to that of Norn [2, 3] can be applied.

Example. Consider the grammar $\mathcal{G} : S \rightarrow \mathtt{a}\ S\ \mathtt{b} \mid S\ \mathtt{b} \mid \epsilon$ containing the start symbol S and two terminals $\mathtt{a}$ and $\mathtt{b}$. Consider the set of constraints $\phi_1 : x \in \mathcal{G}$, $\phi_2 : y \in \mathcal{G}$, $\phi_3 : x = \mathtt{a} \bullet y \bullet z$, and $\phi_4 : x = y \bullet t$, i.e., we have two grammar constraints and two word equations. We apply our CEGAR framework on the example as follows:

Step 1: Over-approximation. We over-approximate the grammar constraints ϕ_1 and ϕ_2 and the equality constraints ϕ_3 and ϕ_4. In the current example, we use the downward closure [10, 44] as an over-approximation scheme for grammars. As mentioned above, other over-approximations such as taking upward closures are also possible to use. Applying the method of [10] transforms G to the regular expression $\mathtt{a}^*\mathtt{b}^*$. Furthermore, the constraints ϕ_3 and ϕ_4 build a cyclic dependency in the sense of [2], since they imply $\mathtt{a}\bullet y\bullet z = y\bullet t$ in which the variable y appears in both sides of the equality. Such a cycle causes existing string solvers such as Norn [3], Z3-str [49], or S3P [42] to run forever. Therefore, we rewrite the equalities using four fresh copies x_1, x_2, y_1 and y_2 of the variables x and y. Thus, we obtain a new set of constraints $\phi'_{1,1} : x_1 \in \mathtt{a}^*\mathtt{b}^*$, $\phi'_{1,2} : x_2 \in \mathtt{a}^*\mathtt{b}^*$ $\phi'_{2,1} : y_1 \in \mathtt{a}^*\mathtt{b}^*$, $\phi'_{2,2} : y_2 \in \mathtt{a}^*\mathtt{b}^*$, $\phi'_3 : x_1 = \mathtt{a} \bullet y_1 \bullet z$, and $\phi'_4 : x_2 = y_2 \bullet t$.

We check the satisfiability of these constraints using one of above mentioned external solvers, and obtain the satisfying assignment v_1 where $v_1(x_1)$ = aa, $v_1(x_2)$ = aa, $v_1(y_1)$ = a, $v_1(y_2)$ = aa, $v_1(z) = \epsilon$, and $v_1(t) = \epsilon$. The strings aa, a, and ϵ can all be generated by flat automata each with one loop whose length is one. Therefore, from v_1 we compute the minimal abstraction parameter $\alpha_1 = \langle \mathbb{p}_1, \mathbb{q}_1 \rangle = \langle 1, 1 \rangle$. We add α_1 to the set Waiting.

Step 2: Under-approximation. Currently, the set Waiting contains one element namely α_1. We fetch α_1 from Waiting and add it to the set Covered. We under-approximate the constraints with $\langle 1, 1 \rangle$. First, we flatten $\mathcal{G}$ with $\langle 1, 1 \rangle$ obtaining $x \in \mathtt{b}^*$ and $y \in \mathtt{b}^*$. The under-approximation is unsatisfiable since, in the flattening of ϕ_3, the string assigned to x must start with a. This will be detected by the procedure since the Parikh image of $\mathtt{b}^*$ captures the fact that we can have any number of b:s and no a:s, while the Parikh image of the flattening of ϕ_3 requires at least one occurrence of a.

Step 3: Over-approximation. We run the over-approximation again, but now we refine it by adding the complement of $x, y \in b^*$, $z, t \in a^* + b^*$. We run a solver again and obtain a new satisfying assignment v_2 where $v_2(x_1)$ = aaab, $v_2(x_2)$ = aaab, $v_2(y_1)$ = aab, $v_2(y_2)$ = aaab, $v_1(z) = \epsilon$, and $v_1(t) = \epsilon$. From v_2, we compute the minimal abstraction parameters $\alpha_2 = \langle \mathbb{p}_2, \mathbb{q}_2 \rangle = \langle 1, 2 \rangle$ and $\alpha_2' = \langle \mathbb{p}_2', \mathbb{q}_2' \rangle = \langle 4, 1 \rangle$, and add them to the set Waiting.

Step 4: Under-approximation. The set Waiting now contains two elements, namely α_2 and α_2'. We select one of them, say α_2, and move it from the set Waiting to the set Covered. We under-approximate the constraints with $\langle 1, 2 \rangle$. When we flatten $\mathcal{G}$ with $\langle 1, 2 \rangle$ we obtain $x \in \mathtt{b}^* + \mathtt{ab}^+$ and $y \in \mathtt{b}^* + \mathtt{ab}^+$. The under-approximation gives the satisfying assignment v_3 where $v_3(x)$ = ab, $v_3(y) = \epsilon$, $v_3(z)$ = b, and $v_3(t)$ = ab. Thus, the procedure terminates after two iterations.

4. Preliminaries

Sets and Strings. We use $\mathbb{N}$ and $\mathbb{Z}$ to denote the sets of natural numbers and integers respectively. For a set A, we use $|A|$ to denote the size of A. Let Σ be a finite alphabet. We use Σ^* to denote the set of finite strings over Σ, and use ϵ to denote the empty string. We define $\Sigma_\epsilon := \Sigma \cup \{\epsilon\}$. For a string $w \in \Sigma^*$, we use $\mathtt{length}(w)$ to denote the length of w. A *language* L over Σ is a set $L \subseteq \Sigma^*$. For strings w and w', we write $w \preceq w'$ to denote that w is a (not necessarily contiguous) substring of w'. For a string $w = a_1 a_2 \cdots a_n \in \Sigma^*$ and $\Sigma' \subseteq \Sigma$, we define $[w]_{\Sigma'}$ to be the largest (not necessarily contiguous) substring $a_{i_1} a_{i_2} \cdots a_{i_m}$ of w such that $a_{i_j} \in \Sigma'$, i.e., we remove from w the elements that are not members of Σ'.

For a set $\mathbb{X}$, an *$\mathbb{X}$-indexed string* over Σ is a mapping $v : \mathbb{X} \mapsto \Sigma^*$, i.e., it assigns to each $x \in \mathbb{X}$, a string $v(x)$ over Σ. An *$\mathbb{X}$-indexed language* K over Σ is a set of $\mathbb{X}$-indexed strings over Σ. For $\mathbb{X}$-indexed languages K_1, K_2, we use $K_1 \cap K_2$ to denote their intersection.

For alphabets Σ_1, Σ_2, a *renaming* from Σ_1 to Σ_2 is a mapping $\mathcal{R} : \Sigma_1 \mapsto \Sigma_2$. For a string $w \in \Sigma_1^*$, we define $\mathcal{R}(w)$ to be the string over Σ_2 we obtain by replacing each symbol a in w by $\mathcal{R}(a)$. For an $\mathbb{X}$-indexed string $v : \mathbb{X} \mapsto \Sigma_1^*$, we define $\mathcal{R}(v) := v'$ where $v'(x) = \mathcal{R}(v(x))$ for all $x \in \mathbb{X}$. For a language L over Σ_1, we define $\mathcal{R}(L) := \{\mathcal{R}(w) \mid w \in L\}$. For an $\mathbb{X}$-indexed language K over Σ_1, we define $\mathcal{R}(K) := \{\mathcal{R}(v) \mid v \in K\}$.

Automata and Grammars. A *Finite-State Automaton* (FSA) is a tuple $\mathcal{A} = \langle Q, \Sigma, \Delta, q_{init}, q_{acc} \rangle$, where Q is a finite set of *states*, Σ is a finite alphabet, $\Delta \subseteq Q \times \Sigma_\epsilon \times Q$ is a finite set of *transitions*, $q_{init} \in Q$ is the *initial state*, and $q_{acc} \in Q$ is the *accepting state*. A *Regular Expression* (RE) $\mathcal{R}$ over Σ is built inductively by including the empty expression $\varnothing$, the members of Σ_ϵ, and closing under union $+$, concatenation $\bullet$, and the Kleene star operator $*$. A *Context-Free Grammar* (CFG) is a quadruple $\mathcal{G} = \langle N, T, P, S \rangle$, where N is a finite set of *non-terminals*, T is a finite set of *terminals*, P is a finite set of *productions*, and $S \in N$ is the *start symbol*. A production $p \in P$ is of the form $A \to \alpha$, where $A \in N$, and $\alpha \in (N \cup T)^*$. We call α the *rhs* of p. The languages $[\![\mathcal{A}]\!]$, $[\![\mathcal{R}]\!]$, $[\![\mathcal{G}]\!]$ of $\mathcal{A}, \mathcal{R}, \mathcal{G}$ are defined in the standard manner.

A *transducer* $\mathcal{T}$ is of the same form as an FSA, the only difference being that now $\Delta \subseteq Q \times \Sigma_\epsilon \times \Sigma_\epsilon \times Q$. For strings $w_1, w_2 \in \Sigma^*$, we write $w_2 \in \mathcal{T}(w_1)$ to denote that there is a sequence $q_0 \langle a_1, b_1 \rangle q_1 \langle a_2, b_2 \rangle \cdots \langle a_n, b_n \rangle q_n$ such that $q_0 = q_{init}$, $q_n = q_{acc}$, $\langle q_i, \langle a_{i+1}, b_{i+1} \rangle, q_{i+1} \rangle \in \Delta$ for all $i : 0 \le i < n$, $w_1 = a_1 a_2 \cdots a_n$, and $w_2 = b_1 b_2 \cdots b_n$.

Presburger Formulas. Presburger arithmetic is the first-order theory of the natural numbers with addition. Here, we introduce a subset of its formulas as follows. A *linear constraint* is of the form $\sum_{1 \le i \le n} k_i \cdot x_i \sim k$ where $k_i \in \mathbb{Z}$ for $i : 1 \le i \le n$, and $k \in \mathbb{Z}$. We define $\mathtt{FV}(\varrho) := \{x_1, \ldots, x_n\}$, i.e., it is the set of (free) variables that occur in ϱ. A *quantifier-free Presburger formula* is a Boolean combination of a set $\{\varrho_1, \ldots, \varrho_n\}$ of linear constraints. We define $\mathtt{FV}(\varrho) := \cup_{1 \le i \le n} \mathtt{FV}(\varrho_i)$. For a valuation $\theta : \mathtt{FV}(\varrho) \mapsto \mathbb{N}$, we write $\theta \vDash \rho$ to denote that ϱ evaluates to true when the linear constraints are evaluated under θ, and the results are combined using the Boolean combinators in ρ. An *existentially quantified Presburger formula* ρ is of the form $\exists y_1 y_2 \cdots y_m . \varrho$ where ϱ is a quantifier-free Presburger formula. For a valuation $\theta : \mathtt{FV}(\varrho) - \{y_1, y_2, \ldots, y_m\} \mapsto \mathbb{N}$, we write $\theta \vDash \varrho$ to denote that there are $a_1, a_2 \ldots, a_m \in \mathbb{N}$ such that $\theta' \vDash \varrho'$ where $\theta'(x) = \theta(x)$ if $x \in \mathtt{FV}(\varrho) - \{y_1, y_2, \ldots, y_m\}$, and $\theta'(x) = a_j$ if $x = y_j$ for some $j : 1 \le j \le m$. We define $[\![\rho]\!] := \{\theta \mid \theta \vDash \rho\}$. Sometimes we use a set notation for the existential quantifiers, and write ρ as $\exists A . \varrho$ where $A = \{y_1, y_2, \ldots, y_m\}$. We assume a function SMT which, given a conjunction $\rho = \rho_1 \wedge \cdots \wedge \rho_n$ of existentially quantified Presburger formulas, either $\mathtt{SMT}(\rho) = \theta$ for some $\theta \in [\![\rho_1]\!] \cap \cdots \cap [\![\rho_n]\!]$, or $\mathtt{SMT}(\rho) = \bot$ if $[\![\rho_1]\!] \cap \cdots \cap [\![\rho_n]\!] = \varnothing$.

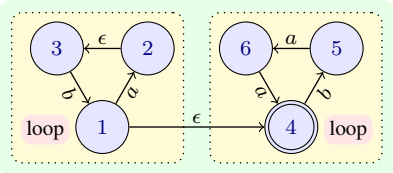

Figure 2. A $\langle 3, 2\rangle$-flat automaton of $(ab)^* \bullet (baa)^*$.

Parikh Images. Consider an alphabet Σ. For a string $w \in \Sigma^*$, we define $\#w : \Sigma \mapsto \mathbb{N}$ to be a function such that, for each symbol $a \in \Sigma$, $\#w\,(a)$ gives the number of occurrences a in w. The Parikh image of a language $L \subseteq \Sigma^*$ is defined by $\#L := \{\#w \mid w \in L\}$. We will characterize the Parikh image of some languages using Presburger formulas. To do that, we define the set $\Sigma^\bullet := \{a^\bullet \mid a \in \Sigma\}$, where $a^\bullet$ is a numerical variable that will be used in the Presburger formula to encode the number of occurrences of a. We say that L is *Parikh-definable* if there is a quantifier-free Presburger formula over $\Sigma^\bullet$, denoted $\texttt{CompP}\,(L)$ (for *Compute Parikh image*), that characterizes the Parikh image of L. More precisely, for any $\theta : \Sigma^\bullet \mapsto \mathbb{N}$, we have $\theta \vDash \texttt{CompP}\,(L)$ iff there is a string $w \in L$ such that $\theta\,(a^\bullet) = \#w\,(a)$ for all symbols $a \in \Sigma$. It is well-known that any context-free (and therefore also any regular) language is Parikh-definable. In fact, given a context-free grammar $\mathcal{G}$, we can compute $\texttt{CompP}\,(\llbracket\mathcal{G}\rrbracket)$ in polynomial time [14, 33]. Notice that this implies that we can also compute $\texttt{CompP}\,(\llbracket\mathcal{R}\rrbracket)$ for a regular expression $\mathcal{R}$ in polynomial time. For simplicity, we sometime write $\texttt{CompP}\,(\mathcal{G})$ and $\texttt{CompP}\,(\mathcal{R})$ instead of $\texttt{CompP}\,(\llbracket\mathcal{G}\rrbracket)$ and $\texttt{CompP}\,(\llbracket\mathcal{R}\rrbracket)$.

We extend the notion of Parikh images to indexed languages as follows. For an indexed string $v : \mathbb{X} \mapsto \Sigma^*$, we define the mapping $\#v : \mathbb{X} \mapsto \Sigma \mapsto \mathbb{N}$ such that, for each variable $x \in \mathbb{X}$ and symbol $a \in \Sigma$, $\#v(x)(a)$ gives the number of occurrences of a in $v\,(x)$. The Parikh image of an $\mathbb{X}$-indexed language K over Σ is defined by $\#K := \{\#v \mid v \in K\}$. We consider the set $(\mathbb{X} \times \Sigma)^\bullet := \left\{\langle x, a\rangle^\bullet \mid (x \in \mathbb{X}) \wedge (a \in \Sigma)\right\}$, and say that K is *Parikh-definable* if there is a quantifier-free Presburger formula over $(\mathbb{X} \times \Sigma)^\bullet$ such that, for any $\theta : (\mathbb{X} \times \Sigma)^\bullet \mapsto \mathbb{N}$, we have $\theta \vDash \texttt{CompP}\,(K)$ iff there is an $\mathbb{X}$-indexed string $w \in K$ such that $\theta\left(\langle x, a\rangle^\bullet\right) = \#v(x)(a)$ for all variables $x \in \mathbb{X}$ and symbols $a \in \Sigma$.

5. String Constraints

Fix a finite alphabet Σ and a finite set of variables $\mathbb{X}$ ranging over Σ^*. Below we define a set of *string constraints* over Σ and $\mathbb{X}$. Each constraint ϕ characterizes an $\mathbb{X}$-indexed language $\llbracket\phi\rrbracket$ over Σ. The set of *terms* $\texttt{Terms}\,(\Sigma, \mathbb{X})$ over Σ and $\mathbb{X}$ is the smallest set such that (i) $(\Sigma \cup \mathbb{X} \cup \{\epsilon\}) \subseteq \texttt{Terms}\,(\Sigma, \mathbb{X})$, and (ii) if $t_1, t_2 \in \texttt{Terms}\,(\Sigma, \mathbb{X})$ then $t_1 \bullet t_2 \in \texttt{Terms}\,(\Sigma, \mathbb{X})$. Given an $\mathbb{X}$-indexed string $v : \mathbb{X} \mapsto \Sigma^*$ over Σ, we extend it to terms by defining $v : \texttt{Terms}\,(\Sigma, \mathbb{X}) \mapsto \Sigma^*$ with $v\,(a) := a$ if $a \in \Sigma$, and $v\,(t_1 \bullet t_2) := v\,(t_1) \bullet v\,(t_2)$.

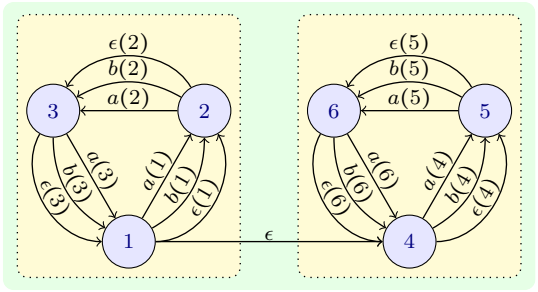

Figure 3. The generic $\langle 3, 2\rangle$-flat automaton.

An *equality constraint* (also called a *word equation*) ϕ is of the form $t_1 = t_2$ where $t_1, t_2 \in \texttt{Terms}\,(\Sigma, \mathbb{X})$. We define $\llbracket\phi\rrbracket := \{v \mid v\,(t_1) = v\,(t_2)\}$. A *disequality constraint* is of the form $t_1 \neq t_2$ and is interpreted analogously.

A *transducer constraint* ϕ is of the form $y \in \mathcal{T}\,(x)$ where $x, y \in \mathbb{X}$ and $\mathcal{T}$ is a transducer. We define $\llbracket\phi\rrbracket := \{v \mid v\,(y) \in \mathcal{T}\,(v\,(x))\}$.

A *grammar constraint* ϕ is of the form $x \in \mathcal{G}$, where $x \in \mathbb{X}$ and $\mathcal{G} = \langle N, T, P, S\rangle$ is a CFG with $T = \Sigma$. We define $\llbracket\phi\rrbracket := \{v \mid v\,(x) \in \llbracket\mathcal{G}\rrbracket\}$. The special case of *regular constraints*, of the form $x \in \mathcal{R}$ where $\mathcal{R}$ is a regular expression over Σ is interpreted in a similar manner.

A *length constraint* ϕ is of the form $\sum_{1\leq i\leq n} k_i \cdot \texttt{length}\,(x_i) \sim k$, where $\sim \in \{<, \leq, >, \geq, =\}$, $x_i \in \mathbb{X}$, and $k_i, k \in \mathbb{Z}$. We define $\llbracket\phi\rrbracket := \{v \mid \sum_{1\leq i\leq n} k_i \cdot \texttt{length}\,(v\,(x_i)) \sim k\}$.

A *string constraint* is of one of the above forms. A *set* ψ of constraints is interpreted as $\llbracket\psi\rrbracket := \cap_{\phi\in\psi} \llbracket\phi\rrbracket$.

6. Flat Languages

In this section, we define flat languages and flat FSAs. The languages will be defined over an alphabet Σ, and their forms will be decided by two parameters $\mathbb{p}, \mathbb{q} \in \mathbb{N}$. For the rest of the section, we fix Σ, $\mathbb{p}$, and $\mathbb{q}$. We define $\alpha := \langle \mathbb{p}, \mathbb{q}\rangle$, and call α the *abstraction parameter*. We introduce generic automata that recognize whole classes of flat languages.

6.1 Flat Languages

A language L over Σ is said to be α-*flat* if there are strings $w_1, w_2, \ldots, w_{\mathbb{q}} \in \Sigma^*$ such that $\texttt{length}\,(w_i) \leq \mathbb{p}$, for each $i : 1 \leq i \leq \mathbb{q}$, and $L = (w_1)^* \bullet (w_2)^* \bullet \cdots \bullet (w_{\mathbb{q}})^*$. We call $w_1, w_2, \ldots, w_n$ the *loops* of L. We can recognize an α-flat language over Σ using a special class of automata, which we call α-*flat automata*. A $\langle 3, 2\rangle$-flat automaton is shown in Fig. 2. The automaton recognizes the $\langle 3, 2\rangle$-flat language $(ab)^* \bullet (baa)^*$. The automaton contains two loops (cycles), each with three states. Below, we define formally the notion of an α-*flat automaton*. We define $\overline{\mathcal{S}}\,(\alpha) := \mathbb{p} \cdot \mathbb{q}$, to give the number of states in the automaton ($\mathbb{q}$ loops each with $\mathbb{p}$ states), and define $\mathcal{S}\,(\alpha) := \left\{i \mid 1 \leq i \leq \overline{\mathcal{S}}\,(\alpha)\right\}$, i.e., we enumerate the states from 1 to $\overline{\mathcal{S}}\,(\alpha)$. We define the set $\texttt{Entries}\,(\alpha) := \{i \mid (i \in \mathcal{S}\,(\alpha)) \wedge (i \mod \mathbb{p} = 1)\}$ which gives the states that are entries of loops in the automaton;

$\texttt{LastEntry}(\alpha) := \overline{\mathcal{S}}(\alpha) - (\mathbb{p} - 1)$, which gives the entry of the last loop in the automaton. We introduce functions that give different types of successors of a state i. Consider $i \in \mathcal{S}(\alpha)$. We define $\texttt{LoopSucc}(\alpha)(i) := \{i+1\}$ if $i \bmod \mathbb{p} \neq 0$, and $\texttt{LoopSucc}(\alpha)(i) := \{i - \mathbb{p} + 1\}$ if $i \bmod \mathbb{p} = 0$, i.e., the function gives the (single) successor of the state that lies within the same loop. We define $\texttt{EntrySucc}(\alpha)(i) := \{i + \mathbb{p}\}$ if $i \in \texttt{Entries}(\alpha) - \{\texttt{LastEntry}(\alpha)\}$, and $\texttt{EntrySucc}(\alpha)(i) := \varnothing$ otherwise, i.e., for a loop entry, the function gives the next loop entry. We define $\texttt{succ}(\alpha)(i) := \texttt{LoopSucc}(\alpha)(i) \cup \texttt{EntrySucc}(\alpha)(i)$. Notice that, if i is the entry of a loop (except the last one) then it has two successors, otherwise it has a single successor. We use $\texttt{succ}^*(\alpha)(\cdot)$ to denote the reflexive transitive closure of $\texttt{succ}(\alpha)(\cdot)$. For example, we have $\overline{\mathcal{S}}(3,2) = 6$, $\texttt{Entries}(3,2) = \{1,4\}$, $\texttt{LastEntry}(3,2) = 4$, $\texttt{LoopSucc}(3,2)(1) = \{2\}$, $\texttt{EntrySucc}(3,2)(1) = \{4\}$, $\texttt{succ}(3,2)(1) = \{2,4\}$, $\texttt{succ}(3,2)(2) = \{3\}$, and $\texttt{succ}^*(3,2)(4) = \{4,5,6\}$. Formally, an α-flat automaton $\mathcal{A}$ is a tuple $\langle Q, \Sigma, \Delta, q_{init}, q_{acc}\rangle$, where (i) $Q = \mathcal{S}(\alpha)$. (ii) $\Delta = \Delta' \cup \Delta''$. The set Δ' contains for each i, j with $j \in \texttt{LoopSucc}(\alpha)(i)$, one (and only one) transition of the form $\langle i, a, j\rangle$ where $a \in \Sigma_\epsilon$. The set Δ'' contains for each i, j with $j \in \texttt{EntrySucc}(\alpha)(i)$, one (and only one) transition of the form $\langle i, \epsilon, j\rangle$, i.e., transitions between two consecutive loop entries are always labeled with ϵ. (iii) $q_{init} = 1$, and (iv) $q_{acc} = \texttt{LastEntry}(\alpha)$, i.e., the accepting state is the entry of the last loop. Notice that, for a given parameter α, all α-flat automata have the same structure, i.e., they are of the same form except that they may differ on the labels of the transitions inside the loops. Also, notice that, since we allow ϵ-transitions, we essentially allow loops of sizes up to $\mathbb{p}$ (rather than equal to $\mathbb{p}$), and allow up to $\mathbb{q}$ loops (rather than exactly $\mathbb{q}$ loops). Given $\mathbb{p}$ and $\mathbb{q}$, there are $|\Sigma_\epsilon|^{\mathbb{p}\cdot\mathbb{q}}$ different α-flat automata over Σ (since each such an automaton contains $\mathbb{p}\cdot\mathbb{q}$ transitions inside its loops, each of which may be labeled by some element in Σ_ϵ). We define the *complete α-flat language over* Σ, by $\mathbb{F}(\alpha) := \bigcup\{L \mid L \text{ is an } \alpha\text{-flat language over } \Sigma\}$, i.e., it is the union of all α-flat languages over Σ. For a set $\mathbb{X}$ of variables, we define the *complete* $\mathbb{X}$-indexed α-flat language over Σ by $\mathbb{F}^{\mathbb{X}}(\alpha) := \{v : \mathbb{X} \mapsto \Sigma^* \mid \forall x \in \mathbb{X}. v(x) \in \mathbb{F}(\alpha)\}$, i.e., it is the set of $\mathbb{X}$-indexed strings over Σ such that each variable is mapped to a string in $\mathbb{F}(\alpha)$.

6.2 Generic Flat Automata

Given the identical structure of all α-flat automata (for a given value of α), we will define a generic automaton that collects the behaviors of all such automata in one.

We will consider the alphabet $\Sigma(\alpha) := \{a(i) \mid (a \in \Sigma_\epsilon) \wedge (i \in \mathcal{S}(\alpha))\} \cup \{\epsilon\}$. We define the *generic* α-flat automaton over Σ, $\mathcal{B}(\alpha) := \langle Q, \Sigma(\alpha), \Delta, q_{init}, q_{acc}\rangle$ (Fig. 3), where Q, q_{init}, and q_{acc} are of the same form as for α-flat automata, and $\Delta = \Delta' \cup \Delta''$ with $\Delta' = \{\langle i, a(i), j\rangle \mid (a \in \Sigma_\epsilon) \wedge (j \in \texttt{LoopSucc}(\alpha)(i))\}$, and $\Delta'' = \{\langle i, \epsilon, j\rangle \mid j \in \texttt{EntrySucc}(\alpha)(i)\}$. In other words, for each state i inside a loop and each symbol $a \in \Sigma$, we add a transition, labeled with $a(i)$ to the next state in the loop. In addition, we put back the ϵ-transitions between the consecutive loop entries. We define the α-generic language $\mathbb{G}(\alpha) := [\![\mathcal{B}(\alpha)]\!]$, i.e., it is the language of $\mathcal{B}(\alpha)$. A string w over $\Sigma(\alpha)$ is said to be *α-generic* (or simply *generic*) if $w \in \mathbb{G}(\alpha)$. The generic α-flat automaton encodes the behaviors of all α-flat automata. More precisely, given an α-flat automaton $\mathcal{A}$, then traversing a transition labeled with (say) a between the two states i and j in a loop, can be simulated by taking the transition labeled with $a(i)$ in the generic automaton. However, the generic automaton also contains additional behaviors that are not exhibited by any individual flat automaton. The reason is that transitions labeled by *different* symbols may be chosen between the same pair of states inside a loop during a single run of the generic automaton. We define the $\mathbb{X}$-indexed language $\mathbb{G}^{\mathbb{X}}(\alpha) := \left\{v : \mathbb{X} \mapsto (\Sigma(\alpha))^* \mid \forall x \in \mathbb{X}.\ v(x) \in \mathbb{G}(\alpha)\right\}$. An $\mathbb{X}$-indexed string v is *(α-)generic* if $v \in \mathbb{G}^{\mathbb{X}}(\alpha)$.

To avoid the problem of choosing different symbols between identical pairs of states, we will intersect the language of a generic automaton with a language whose words encode a *purity* condition, in the sense they guarantee that at most one outgoing transition of each state is chosen during the iterations of the loops in the automaton. Formally, for a string $w \in (\Sigma(\alpha))^*$ we say that w is *pure* if for all $i \in \mathcal{S}(\alpha)$ and all $a, b \in \Sigma$ with $a \neq b$, it is the case that $\#w(a(i)) > 0$ implies $\#w(b(i)) = 0$. An indexed string $v : \mathbb{X} \mapsto (\Sigma(\alpha))^*$ is said to be *pure* if $v(x)$ is pure for all $x \in \mathbb{X}$. We define the language $\mathbb{P}^{\mathbb{X}}(\alpha) := \left\{v : \mathbb{X} \mapsto (\Sigma(\alpha))^* \mid v \text{ is pure}\right\}$. A $\mathbb{X}$-indexed language K over $\Sigma(\alpha)$ is said to be *(α-)generic* if $K \subseteq \mathbb{G}^{\mathbb{X}}(\alpha)$, and it is called *pure* if $K \subseteq \mathbb{P}^{\mathbb{X}}(\alpha)$.

LEMMA 1. *For $\mathbb{X}$-indexed strings $v_1, v_2 \in (\mathbb{G}^{\mathbb{X}}(\alpha) \cap \mathbb{P}^{\mathbb{X}}(\alpha))$, if $(\#v_1) = (\#v_2)$ then $v_1 = v_2$.*

Lemma 1 follows from the fact that if $(\#v_1) = (\#v_2)$ then v_1 and v_2 correspond to identical runs of the generic automaton on each variable, i.e., the loops are iterated an identical number of times, and, for each state, the same outgoing transition is chosen inside the relevant loop.

Lemma 1 allows us to define a partial function $\texttt{GetS}$ such that for any $\theta : (\mathbb{X} \times \Sigma(\alpha))^\bullet \mapsto \mathbb{N}$, the value of $\texttt{GetS}(\theta)$ is the unique $\mathbb{X}$-string $v \in \mathbb{G}^{\mathbb{X}}(\alpha) \cap \mathbb{P}^{\mathbb{X}}(\alpha)$ with $\theta\left(\langle x, a(i)\rangle^\bullet\right) = \#v(x)(a(i))$ for all variables $x \in \mathbb{X}$, symbols $a \in \Sigma$, and $i : 1 \leq i \leq \overline{\mathcal{S}}(\alpha)$. Notice that $\texttt{GetS}(\theta)$ may not exist. However, if it exists then, by Lemma 1, it is unique. We get the following Corollary.

COROLLARY 1. *For an $\mathbb{X}$-indexed language $K \subseteq (\mathbb{G}^{\mathbb{X}}(\alpha) \cap \mathbb{P}^{\mathbb{X}}(\alpha))$, if $\theta \in \#K$ then $\texttt{GetS}(\theta) \in K$.*

Also, Lemma 1 implies the following lemma.

LEMMA 2. *For $\mathbb{X}$-indexed languages $K_1, K_2 \subseteq (\mathbb{G}^{\mathbb{X}}(\alpha) \cap \mathbb{P}^{\mathbb{X}}(\alpha))$, it is the case that $(\#K_1) \cap (\#K_2) = \#(K_1 \cap K_2)$.*

Informally, for two pure and generic languages, the Parikh images of their intersection can be computed by computing the Parikh images individually and taking the intersection.

7. Flattening

Fix a set of variables $\mathbb{X}$, an alphabet Σ, parameters $\mathbb{p}, \mathbb{q} \in \mathbb{N}$, and $\alpha = \langle \mathbb{p}, \mathbb{q} \rangle$. We will describe how to construct the *flattening* of a string constraint ϕ. The constraint ϕ may be in of the forms described in Section 5. The flattening of ϕ corresponds to taking the intersection of $[\![\phi]\!]$ with the generic α-flat automaton. We will take the flattening of ϕ and intersect it with the set of pure languages thus obtaining a particular $\mathbb{X}$-indexed language $\langle\!\langle \phi \rangle\!\rangle_\alpha$ that satisfies two important properties. First, $\langle\!\langle \phi \rangle\!\rangle_\alpha$ characterizes the intersection of ϕ and flat languages in the sense that any indexed string in $\langle\!\langle \phi \rangle\!\rangle_\alpha$ can be renamed to an indexed string that is in the intersection of $[\![\phi]\!]$ and flat languages. (ii) $\langle\!\langle \phi \rangle\!\rangle_\alpha$ is Parikh definable (see Section 8.)

7.1 Flattening Grammar Constraints

Consider a grammar constraint ϕ of the form $x \in \mathcal{G}$ with $\mathcal{G} = \langle N, T, P, S \rangle$, $T = \Sigma$, and a parameter $\alpha = \langle \mathbb{p}, \mathbb{q} \rangle$. We will define a new grammar $\texttt{Flatten}(\alpha)(\phi)$ which encodes running $\mathcal{G}$ "in parallel" with the α-generic automaton. We define $\texttt{Flatten}(\alpha)(\phi) := \langle N', T', P', S' \rangle$, where $T' := \Sigma(\alpha)$, and define the set $N' := N'_1 \cup N'_2 \cup N'_3$ as the union of three sets of nonterminals:

- For each nonterminal $A \in N$ and $i, j \in \mathcal{S}(\alpha)$ with $j \in \texttt{succ}^*(\alpha)(i)$, the set N'_1 contains a corresponding nonterminal $A^{\oplus}(i,j)$, i.e., $N'_1 := \{A^{\oplus}(i,j) \mid (A \in N) \wedge (j \in \texttt{succ}^*(\alpha)(i))\}$. Intuitively, the next segment of the input string expected by $\mathcal{G}$ corresponds to A, while the automaton is currently in state i. We use $A^{\oplus}$ to allow the automaton to perform a number of transitions to consume the same part of the input string, after which the automaton reaches state j.
- For each $a \in T$ and $i, j \in \mathcal{S}(\alpha)$ with $j \in \texttt{succ}^*(\alpha)(i)$, the set N'_2 contains a corresponding nonterminal $a^{\oplus}(i,j)$, i.e., $N'_2 := \{a^{\oplus}(i,j) \mid (a \in T) \wedge (j \in \texttt{succ}^*(\alpha)(i))\}$. Intuitively, the next terminal expected by $\mathcal{G}$ is a. The automaton is currently in the state i, and may perform an arbitrary number of ϵ-transitions both before and after performing a transition labeled with a, ending up in the state j.
- For each $i, j \in \mathcal{S}(\alpha)$ with $j \in \texttt{succ}^*(\alpha)(i)$, the set N'_3 contains a corresponding nonterminal $\epsilon^{\oplus}(i,j)$, i.e., $N'_3 := \{\epsilon^{\oplus}(i,j) \mid j \in \texttt{succ}^*(\alpha)(i)\}$. This allows the automaton to perform an arbitrary number of ϵ-transitions.

We define the start symbol $S' := S(1, \texttt{LastEntry}(\alpha))$, and define the set $P' := P'_1 \cup P'_2 \cup P'_3 \cup P'_4$ as follows:

- For each production $p \in P$ of the form $A \to X_1 \cdot X_2 \cdots X_n$, and $i, j \in \mathcal{S}(\alpha)$ with $j \in \texttt{succ}^*(\alpha)(i)$, the set P'_1 contains all productions of the form $A^{\oplus}(i,j) \to X_1^{\oplus}(i_0, i_1) \cdot X_2^{\oplus}(i_1, i_2) \cdots X_n^{\oplus}(i_{n-1}, i_n)$, where $i_0 = i$, $i_n = j$, $i_k \in \texttt{succ}^*(\alpha)(i_{k-1})$, for $k : 1 \le k \le n$. The next segment of the input string can be consumed by $\mathcal{G}$ and (in parallel) by the automaton. This is done by dividing the segment into sub-segments according to the *rhs* of p, by letting $\mathcal{G}$ and the automaton run in parallel on each sub-segment.
- For each terminal $a \in T$ and $i, j \in \mathcal{S}(\alpha)$ with $j \in \texttt{succ}^*(\alpha)(i)$, the set P'_2 contains all productions of the form $a^{\oplus}(i,j) \to \epsilon^{\oplus}(i_0, i_1) \cdot a(i_1) \cdot \epsilon^{\oplus}(i_2, i_3)$, where $i_0 = i$, $i_3 = j$, $i_1 \in \texttt{succ}^*(\alpha)(i_0)$, $i_2 \in \texttt{LoopSucc}(\alpha)(i_1)$, and $i_3 \in \texttt{succ}^*(\alpha)(i_2)$. The automaton is allowed to perform an arbitrary number of ϵ-transitions, before and after a transition labeled by a. The latter is part of a loop.
- The set P'_3 contains the following sets of productions (that allow the automaton to perform an arbitrary number of ϵ-transitions)
  - All productions that are of the form $\epsilon^{\oplus}(i,j) \to \epsilon(i) \cdot \epsilon^{\oplus}(k,j)$, where $i, j, k \in \mathcal{S}(\alpha)$, $k \in \texttt{LoopSucc}(\alpha)(i)$, and $j \in \texttt{succ}^*(\alpha)(k)$, i.e., the automaton performs one ϵ-transition from the state i and then takes some number of ϵ-transitions to the state j.
  - All productions that are of the form $\epsilon^{\oplus}(i,i) \to \epsilon$, where $i \in \mathcal{S}(\alpha)$, i.e., stop generating ϵ-transitions.
- For all $i, j \in \mathcal{S}(\alpha)$ with $j \in \texttt{EntrySucc}(\alpha)(i)$, the set P'_4 contains the production $A(i,j) \to \epsilon$. The automaton is allowed to cross from one loop entry to the next.

We define $\langle\!\langle \phi \rangle\!\rangle_\alpha$ to be the $\mathbb{X}$-indexed language over $\Sigma(\alpha)$ such that $v \in \langle\!\langle \phi \rangle\!\rangle_\alpha$ iff (i) v is pure, (ii) $v(x) \in [\![\texttt{Flatten}(\alpha)(\phi)]\!]$, and (iii) $v(y) \in \mathbb{G}(\alpha)$ for all $y \in \mathbb{X} - \{x\}$. Intuitively, the variable x is mapped to a pure string in the language of $\mathcal{G}$, while any other variable is mapped to any pure α- generic string. Notice that $\langle\!\langle \phi \rangle\!\rangle_\alpha$ is pure and α-generic.

7.2 Flattening Equality Constraints

We consider an equality constraint ϕ of the form $x_1 x_2 \cdots x_m = x_{m+1} x_{m+2} \cdots x_n$, and a parameter $\alpha = \langle \mathbb{p}, \mathbb{q} \rangle$. A constant c in an equality constraint can be replaced by a fresh variable x with a regular constraint $x \in [\![c]\!]$. Therefore, we assume, without loss of generality, that equality constraints do not contain any constants. We will define an FSA $\texttt{Flatten}(\alpha)(\phi)$ that will run the concatenation of the generic flat automata for the variables $x_1, x_2, \ldots, x_m$, in parallel with the concatenation of the generic flat automata for $x_{m+1}, x_{m+2}, \ldots, x_n$. Essentially, it traverses the product of the two concatenations, and enforces synchronization on common alphabet symbols. We define the alphabet $\Sigma(n, \alpha) := \{a(k,i) \mid (a \in \Sigma_\epsilon) \wedge (1 \le k \le n) \wedge (i \in \mathcal{S}(\alpha))\}$. We define $\texttt{Flatten}(\alpha)(\phi) := \langle Q, \Sigma(n, \alpha), \Delta, q_{init}, q_{acc} \rangle$

as follows. We define the set $Q := Q_1 \cup Q_2$ as the union of two sets of states:

- For each $k : 1 \leq k \leq m$, $\ell : m+1 \leq \ell \leq n$, and $i_1, i_2 \in \mathcal{S}(\alpha)$, the set Q_1 contains the state $\langle k, i_1, \ell, i_2 \rangle$. Each state in Q_1 encodes (i) an index k showing which automaton, among the ones of $x_1, x_2, \ldots x_m$, we are currently simulating, (ii) the current state i_1 of that automaton, (iii) an index ℓ showing which automaton, among the ones of $x_{m+1}, x_{m+2}, \ldots x_n$, we are currently simulating, and (iv) the current state i_2 of that automaton.
- For each $k : 1 \leq k \leq m$, $\ell : m+1 \leq \ell \leq n$, $i_1, i_2 \in \mathcal{S}(\alpha)$, and $a \in \Sigma$, the set Q_2 contains the state $\langle k, i_1, \ell, i_2, a \rangle$. This state encodes that the automaton of x_k has just performed a transition labeled by a. The automaton of x_ℓ will follow by performing a transition labeled by a.

We define the set $\Delta := \Delta_1 \cup \Delta_2 \cup \Delta_3 \cup \Delta_4 \cup \Delta_5 \cup \Delta_6 \cup \Delta_7 \cup \Delta_8$ as the union of eight sets of transitions:

- For each $a \in \Sigma$, and k, i_1, ℓ, i_2, j with $1 \leq k \leq m$, $m+1 \leq \ell \leq n$, $i_1, i_2, j \in \mathcal{S}(\alpha)$, and $j \in \texttt{LoopSucc}(\alpha)(i_1)$, the set Δ_1 contains the transition $\langle \langle k, i_1, \ell, i_2 \rangle, a(k, i_1), \langle k, j, \ell, i_2, a \rangle \rangle$. This corresponds to the case where the automaton of x_k performs a transition labeled with a.
- For each $a \in \Sigma$, and k, i_1, ℓ, i_2, j with $1 \leq k \leq m$, $m+1 \leq \ell \leq n$, $i_1, i_2, j \in \mathcal{S}(\alpha)$, and $j \in \texttt{LoopSucc}(\alpha)(i_2)$, the set Δ_2 contains the transition $\langle \langle k, i_1, \ell, i_2, a \rangle, a(\ell, i_2), \langle k, i_1, \ell, j \rangle \rangle$. This corresponds to the case where the automaton of x_ℓ performs a transition labeled with a (answering the previous move of the automaton of x_k).
- For each k, i_1, ℓ, i_2, j with $1 \leq k \leq m$, $m+1 \leq \ell \leq n$, $i_1, i_2, j \in \mathcal{S}(\alpha)$, and $j \in \texttt{LoopSucc}(\alpha)(i_1)$, the set Δ_3 contains the transition $\langle \langle k, i_1, \ell, i_2 \rangle, \epsilon(k, i_1), \langle k, j, \ell, i_2 \rangle \rangle$. This corresponds to the case where the automaton of x_k makes an ϵ-transition, while the automaton of x_ℓ does not move.
- For each k, i_1, ℓ, i_2, j with $1 \leq k \leq m$, $m+1 \leq \ell \leq n$, $i_1, i_2, j \in \mathcal{S}(\alpha)$, and $j \in \texttt{LoopSucc}(\alpha)(i_2)$, the set Δ_4 contains the transition $\langle \langle k, i_1, \ell, i_2 \rangle, \epsilon(\ell, i_2), \langle k, i_1, \ell, j \rangle \rangle$. This case is symmetric to the previous one.
- For each k, i_1, ℓ, i_2, j with $1 \leq k \leq m$, $m+1 \leq \ell \leq n$, $i_1, i_2, j \in \mathcal{S}(\alpha)$, and $j \in \texttt{EntrySucc}(\alpha)(i_1)$, the set Δ_5 contains the transition $\langle \langle k, i_1, \ell, i_2 \rangle, \epsilon, \langle k, j, \ell, i_2 \rangle \rangle$. This corresponds to the case where the automaton of x_k crosses from one loop entry to the next.
- For each k, i_1, ℓ, i_2, j with $1 \leq k \leq m$, $m+1 \leq \ell \leq n$, $i_1, i_2, j \in \mathcal{S}(\alpha)$, and $j \in \texttt{EntrySucc}(\alpha)(i_2)$, the set Δ_6 contains the transition $\langle \langle k, i_1, \ell, i_2 \rangle, \epsilon, \langle k, i_1, \ell, j \rangle \rangle$. This case is symmetric to the previous one.
- For each k, i_1, ℓ, i_2, j with $1 \leq k < m$, $m+1 \leq \ell \leq n$, $i_1, i_2, j \in \mathcal{S}(\alpha)$, $i_1 = \texttt{LastEntry}(\alpha)$, and $j = 1$, the set Δ_7 contains the transition $\langle \langle k, i_1, \ell, i_2 \rangle, \epsilon, \langle k+1, j, \ell, i_2 \rangle \rangle$. The automaton of x_k is in its accepting state; the simulation continues from the initial state of the automaton of x_{k+1}. The automaton of x_ℓ does not move.
- For each k, i_1, ℓ, i_2, j with $1 \leq k \leq m$, $m+1 \leq \ell < n$, $i_1, i_2, j \in \mathcal{S}(\alpha)$, and $i_2 = \texttt{LastEntry}(\alpha)$, the set Δ_8 contains the transition $\langle \langle k, i_1, \ell, i_2 \rangle, \epsilon, \langle k, i_1, \ell+1, j \rangle \rangle$. This case is symmetric to the previous one.

We define the initial state as $q_{init} := \langle 1, 1, 1, 1 \rangle$, i.e., we start from the initial state of the automaton of x_1, and the initial state of the automaton of x_{m+1}. We define the accepting state as $q_{acc} := \langle m, \texttt{LastEntry}(\alpha), n, \texttt{LastEntry}(\alpha) \rangle$, i.e., we are in the accepting states (i.e., last loop entries) of the automata of x_m and x_n respectively.

To derive the indexed language $\langle\!\langle \phi \rangle\!\rangle_\alpha$ we need to give some definitions. First we formulate some conditions on the strings generated by $\texttt{Flatten}(\alpha)(\phi)$. A string $w \in (\Sigma(n, \alpha))^*$ is said to be *rational* if $\#a(k,i) = \#a(\ell,i)$ whenever $x_k = x_\ell$. In other words, different occurrences of the same variable will run the corresponding generic automaton in the same manner (it picks the same outgoing transition from each state and runs the same loop an identical number of times.) We say that w is *pure* if $a \neq b$ and $\#w(a(k,i)) > 0$ implies $\#w(b(k,i)) = 0$. We will use the purity of strings in $\texttt{Flatten}(\alpha)(\phi)$ to guarantee the purity of $\langle\!\langle \phi \rangle\!\rangle_\alpha$. Next, we take the strings generated by $\texttt{Flatten}(\alpha)(\phi)$ and project them on the variables that occur in ϕ. For a $k : 1 \leq k \leq n$, we define the alphabet $\Sigma_k := \{a(k,i) \mid (a \in \Sigma) \wedge (1 \leq i \leq \mathcal{S}(\alpha))\}$, i.e., it is the subset of $\Sigma(n, \alpha)$ containing only the elements in the alphabet of the generic flat automaton of x_k. We define the renaming $\mathcal{R} : \Sigma(n, \alpha) \mapsto \Sigma(\alpha)$ such that $\mathcal{R}(a(k,i)) = a(i)$.

The language $\langle\!\langle \phi \rangle\!\rangle_\alpha$ contains all $\mathbb{X}$-indexed strings $v : \mathbb{X} \mapsto \Sigma(\alpha)$ such that there is a string $w \in \texttt{Flatten}(\alpha)(\phi)$ satisfying the following properties: (i) w is rational and pure. (ii) $v(x) = \mathcal{R}\left([w]_{\Sigma_k}\right)$ if $x = x_k$ for some $k : 1 \leq k \leq n$. In other words, we extract the substring of w corresponding to x_k and rename it according to $\mathcal{R}$ so that we obtain a string over $\Sigma(\alpha)$. Notice that by the rationality condition, the particular choice of k is not important (we can choose any k provided that $x_k = x$.) Also, observe that $\mathcal{R}\left([w]_{\Sigma_k}\right)$ is α-generic. (iii) $v(x) \in \mathbb{G}(\alpha) \cap \mathbb{P}(\alpha)$ if $x \in \mathbb{X} - \{x_1, \ldots, x_n\}$. Such a variable is not restricted by ϕ and hence it may be assigned any pure string in the generic α-flat automaton. Notice that $\langle\!\langle \phi \rangle\!\rangle_\alpha$ is pure and α-generic.

7.3 Other Constraints

The flattening of a transducer constraint $y \in \mathcal{T}(x)$ is done by constructing a FSA that runs $\mathcal{T}$ in parallel with the flat automata of x and y. The construction is similar to the case of equality constraints. A disequality constraint can be

done in a similar way as in the case of equality constraints. In contrast, here we make that eventually one side cannot follow the other. Finally, flattening is not needed for the case of length constraints.

7.4 Properties

Lemma 3 and Lemma 4 below follow from the flattening construction. They explain the relation between $\langle\!\langle \phi \rangle\!\rangle_\alpha$ and the intersection of ϕ with flat languages. Define the renaming $\mathcal{R}^\alpha : \Sigma(\alpha) \mapsto \Sigma$ where $\mathcal{R}^\alpha(a(i)) = a$ for all $a \in \Sigma$ and $i : 1 \leq i \leq \overline{\mathcal{S}}(\alpha)$.

LEMMA 3. $\mathcal{R}^\alpha(\langle\!\langle \phi \rangle\!\rangle_\alpha) \subseteq [\![\phi]\!] \cap \mathbb{F}^{\mathbb{X}}(\alpha)$.

LEMMA 4. *$v \in [\![\phi]\!] \cap \mathbb{F}^{\mathbb{X}}(\alpha)$ implies that $v' \in \langle\!\langle \phi \rangle\!\rangle_\alpha$ for all v' with $\mathcal{R}^\alpha(v') = v$.*

For a set ψ of constraints, we define $\langle\!\langle \psi \rangle\!\rangle_\alpha := \bigcap_{\phi \in \psi} \langle\!\langle \phi \rangle\!\rangle_\alpha$. From Lemma 3 and Lemma 4, we get the following theorem.

THEOREM 1. *$\langle\!\langle \psi \rangle\!\rangle_\alpha = \varnothing$ iff $[\![\psi]\!] \cap \mathbb{F}^{\mathbb{X}}(\alpha) = \varnothing$.*

From Corollary 1 and Lemma 3 we get the following.

THEOREM 2. *If $\theta \in \#\langle\!\langle \psi \rangle\!\rangle_\alpha$ then $\mathcal{R}^\alpha(\texttt{GetS}(\theta)) \in [\![\psi]\!] \cap \mathbb{F}^{\mathbb{X}}(\alpha)$.*

8. Under-Approximation

In this section, we describe the under-approximation module. Fix a finite alphabet Σ, and a finite set of variables $\mathbb{X}$ ranging over Σ^*. Suppose that we are given a set ψ of string constraints over $\mathbb{X}$ and Σ, together with an abstraction parameter $\alpha = \langle \mathbb{p}, \mathbb{q} \rangle$. We introduce an algorithm `UAprx` which checks the emptiness of the set $[\![\psi]\!] \cap \mathbb{F}^{\mathbb{X}}(\alpha)$, and returns a member of the set in case the set is non-empty. We define `UAprx` in several steps. By Theorem 1 we know that to check the emptiness of $[\![\psi]\!] \cap \mathbb{F}^{\mathbb{X}}(\alpha)$, it is sufficient to check the emptiness of $\langle\!\langle \psi \rangle\!\rangle_\alpha$. Notice that the emptiness of the latter is equivalent to the emptiness of its Parikh image. Since $\langle\!\langle \psi \rangle\!\rangle_\alpha$ is by construction pure and α-generic for each $\phi \in \psi$, it follows by Lemma 2 that the Parikh image of $\langle\!\langle \psi \rangle\!\rangle_\alpha$ is equal to the intersection of the Parikh images of $\langle\!\langle \phi \rangle\!\rangle_\alpha$ for all $\phi \in \psi$. First, we describe how to compute the Parikh image of $\langle\!\langle \phi \rangle\!\rangle_\alpha$. Then, we collect the Parikh images for all $\phi \in \psi$, and feed them into an SMT solver. If the SMT solver answers that the Parikh image is empty then `UAprx` answers that $[\![\psi]\!] \cap \mathbb{F}^{\mathbb{X}}(\alpha)$ is empty. On the other hand, if the SMT solver returns a satisfying assignment θ then we know by Theorem 2 that $\mathcal{R}^\alpha(\texttt{GetS}(\theta)) \in [\![\psi]\!] \cap \mathbb{F}^{\mathbb{X}}(\alpha)$. Therefore, we will also present a method for computing $\mathcal{R}^\alpha(\texttt{GetS}(\theta))$.

8.1 Computing Parikh Images

We give an algorithm for computing the Parikh image of $\langle\!\langle \phi \rangle\!\rangle_\alpha$ for a string constraint ϕ. The form of the algorithm depends on the type of ϕ. In each case, the Parikh image will be defined as a conjunction of existentially quantified Presburger formulas over the alphabet $(\mathbb{X} \times \Sigma(\alpha))^\bullet$. The algorithms are defined based on the construction of $\langle\!\langle \phi \rangle\!\rangle_\alpha$, described in Section 7. We present the method for the cases of grammars and equalities. The other cases are similar.

Grammars. Algorithm 1 shows the case where ϕ is a grammar constraint (of the form $x \in \mathcal{G}$). We compute the

Algorithm 1: Computing the Parikh Image of a Grammar Constraint.

Input: ϕ: grammar constraint of the form $x \in \mathcal{G}$,
$\alpha = \langle \mathbb{p}, \mathbb{q} \rangle \in \mathbb{N}^2$: abstraction parameter
Output: $\texttt{CompP}(\langle\!\langle \phi \rangle\!\rangle_\alpha)$

1 $\mathcal{G}' \leftarrow \texttt{Flatten}(\alpha)(\phi)$;
2 $\rho_1 \leftarrow \texttt{CompP}(\mathcal{G}')$;
3 $\rho_2 \leftarrow \bigwedge_{i \in \mathcal{S}(\alpha)} \bigwedge_{a \in \Sigma} \langle x, a(i) \rangle^\bullet = (a(i))^\bullet$;
4 $\rho_3 \leftarrow \bigwedge_{i \in \mathcal{S}(\alpha)} \bigwedge_{a \neq b \in \Sigma} (\langle x, a(i) \rangle^\bullet > 0) \implies (\langle x, b(i) \rangle^\bullet = 0)$;
5 $\rho_4 \leftarrow \exists (\Sigma(\alpha))^\bullet .\ \rho_1 \wedge \rho_2 \wedge \rho_3$;
6 **return** (ρ_4)

flattening $\mathcal{G}'$ of the grammar $\mathcal{G}$ wrt. the abstraction parameter α according to the construction of Section 7. We compute the Parikh image of $\mathcal{G}'$ (this is possible since $\mathcal{G}'$ is a CFG), and store the result in ρ_1. Notice that ρ_1 is defined over the set $(\Sigma(\alpha))^\bullet$. We define the formula ρ_2 that renames each variable $(a(i))^\bullet$ to the corresponding variable $\langle x, a(i) \rangle^\bullet$. This is done by equating each pair of variables of the above form, and putting all the equalities in ρ_2. The formula ρ_3 encodes the purity condition. The returned formula ρ_4 is the conjunction of the previous three formulas. Furthermore, we quantify away all the variables in $(\Sigma(\alpha))^\bullet$ thus ensuring that ρ_3 is defined over the alphabet $(\mathbb{X} \times \Sigma(\alpha))^\bullet$. Notice that ρ_4 is an existentially quantified Presburger formula.

Equalities. Algorithm 2 shows the case where ϕ is an equality constraint (of the form $x_1 x_2 \cdots x_m = x_{m+1} x_{m+2} \cdots x_n$). In a similar manner to the case of grammar constraints, we compute the flattening $\mathcal{A}$ of the equality wrt. the abstraction parameter α, and then compute the Parikh image of the automaton $\mathcal{A}$, and store the result in ρ_1. Here, ρ_2 serves *two* purposes. First, it renames the variables

Algorithm 2: Computing the Parikh Image of an Equality Constraint.

Input: ϕ: equality constraint of the form
$x_1 x_2 \cdots x_m = x_{m+1} x_{m+2} \cdots x_n$,
$\alpha = \langle \mathbb{p}, \mathbb{q} \rangle \in \mathbb{N}^2$: abstraction parameter
Output: $\texttt{CompP}(\langle\!\langle \phi \rangle\!\rangle_\alpha)$

1 $\mathcal{A} \leftarrow \texttt{Flatten}(\alpha)(\phi)$;
2 $\rho_1 \leftarrow \texttt{CompP}(\mathcal{A})$;
3 $\rho_2 \leftarrow \bigwedge_{1 \leq k \leq n} \bigwedge_{i \in \mathcal{S}(\alpha)} \bigwedge_{a \in \Sigma} \langle x_k, a(i) \rangle^\bullet = (a(k,i))^\bullet$;
4 $\rho_3 \leftarrow \bigwedge_{i \in \mathcal{S}(\alpha)} \bigwedge_{a \neq b \in \Sigma} (\langle x, a(i) \rangle^\bullet > 0) \implies (\langle x, b(i) \rangle^\bullet = 0)$;
5 $\rho_4 \leftarrow \exists (\Sigma(n, \alpha))^\bullet .\ \rho_1 \wedge \rho_2 \wedge \rho_3$;
6 **return** (ρ_4)

as in the case of a grammar constraint. More precisely, the formula ρ_1 is defined over the set $(\Sigma(n,\alpha))^\bullet$. Therefore, we rename each variable $(a(k,i))^\bullet$ to the corresponding variable $\langle x_k, a(i)\rangle^\bullet$ by equating them. The second purpose is to ensure the rationality condition. The reason is that, for any two variables of the forms $a(k,i)$ and $a(\ell,i)$, the formula ρ_2 will contain both $\langle x_k, a(i)\rangle^\bullet = (a(k,i))^\bullet$ and $\langle x_\ell, a(i)\rangle^\bullet = (a(\ell,i))^\bullet$. This implies that if $x_k = x_\ell$ then $\langle x_k, a(i)\rangle^\bullet = \langle x_\ell, a(i)\rangle^\bullet$. Finally, we add the purity condition and abstract away the variables of the set $(\Sigma(n,\alpha))^\bullet$ in a similar manner to the case of grammar constraints.

8.2 SMT Solving

In Algorithm 3, we are given a set ψ of constraints together with an abstraction parameter α. We construct, for each $\phi \in \psi$, the Parikh image of $\langle\!\langle \phi \rangle\!\rangle_\alpha$ as described in Section 8.1. We collect the conjunction of the Parikh images in ρ. We check the satisfiability of ρ using the available SMT solver.

Algorithm 3: Parikh Image Analysis.

Input: ψ: set of string constraints,
$\alpha = \langle \mathbb{p}, \mathbb{q} \rangle \in \mathbb{N}^2$: abstraction parameter
Output: Solution for $\texttt{CompP}(\langle\!\langle \psi \rangle\!\rangle_\alpha)$.

1 $\rho \leftarrow \bigwedge_{\phi \in \psi} \texttt{CompP}(\langle\!\langle \phi \rangle\!\rangle_\alpha)$;
2 $\texttt{Result} \leftarrow \texttt{SMT}(\rho)$;
3 **return** (Result)

8.3 Constructing a Solution

Algorithm 4 constructs the $\mathbb{X}$-indexed string $v = \mathcal{R}^\alpha(\texttt{GetS}(\theta))$. More precisely, the algorithm goes through

Algorithm 4: Translating to a Solution.

Input: $\theta : (\mathbb{X} \times \Sigma(\alpha))^\bullet \mapsto \mathbb{N}$,
$\alpha = \langle \mathbb{p}, \mathbb{q} \rangle \in \mathbb{N}^2$: abstraction parameter
Output: $\mathcal{R}^\alpha(\texttt{GetS}(\theta))$

1 **for all** $x \in \mathbb{X}$ **do**
2 **for** k **from** 1 **to** $\mathbb{q}$ **do**
3 $n_k \leftarrow 0$;
4 $w_k \leftarrow \epsilon$;
5 **for** i **from** $(k \cdot \mathbb{p} - \mathbb{p} + 1)$ **to** $(k \cdot \mathbb{p})$ **do**
6 **for all** $a \in \Sigma_\epsilon$ **do**
7 **if** $\theta(\langle x, a(i)\rangle^\bullet) > 0$ **then**
8 $n_k \leftarrow \theta(\langle x, a(i)\rangle^\bullet)$;
9 $w_k \leftarrow w_k \bullet a$;
10 $v(x) \leftarrow w_1^{n_1} \bullet w_1^{n_2} \bullet \cdots \bullet w_{\mathbb{q}}^{n_{\mathbb{q}}}$;
11 **return** (v)

the variables one by one. For each variable $x \in \mathbb{X}$ it considers the generic automaton of x and finds out (i) for each loop $k : 1 \le k \le \mathbb{q}$, the number n_k of times the loop is iterated, and (ii) for each state $i : k \cdot \mathbb{p} - \mathbb{p} + 1 \le i \le k \cdot \mathbb{p}$ inside the loop, the label of the outgoing transition that is chosen. In fact, the algorithm builds the string w_k which corresponds to one iteration of the loop. This is done by recording, for each $a \in \Sigma$, the number of times the symbol $a(i)$ is encountered. Recall that either this number is equal to 0 for all $a \in \Sigma$ or positive for exactly one $a \in \Sigma$. In the former case, the loop has not been iterated, and in the latter case, the loop has been iterated the same number of times as the number of occurrences of $a(i)$. We build the string w_k successively, by concatenating the symbol $a(i)$ in position $i - k \cdot \mathbb{p} + \mathbb{p}$ if $a(i)$ occurs a positive number of times. Finally, for a variable $x \in \mathbb{X}$, we define the string $v(x)$ by concatenating the strings $w_k^{n_k}$ for all the loops $k : 1 \le k \le \mathbb{q}$.

9. Over-Approximation

In this section, we describe the over-approximation module. Fix a finite alphabet Σ and a finite set of variables $\mathbb{X}$ ranging over Σ^*. Suppose that we are given a set $\psi = \{\phi_1, \phi_2, \ldots, \phi_k\}$ of constraints together with a set $\texttt{Covered} \subseteq \mathbb{N}^2$ of parameter values that have already been considered by the under-approximation module. In the following, we will construct a set ψ' of constraints such that $(\llbracket \psi \rrbracket - \bigcup_{\alpha \in \texttt{Covered}} \mathbb{F}^{\mathbb{X}}(\alpha)) \subseteq \llbracket \psi' \rrbracket$. To construct ψ' from ψ, we proceed as follows: (1) we replace any membership constraint in a context-free grammar $\mathcal{G}$ by a membership constraint in a regular language (the regular language may be the upward closure of the language of $\mathcal{G}$ [4, 45], the downward closure [10, 44], or some other over-approximation, e.g., the one in [32]), (2) we replace a transducer constraint by a conjunction of membership constraints in regular languages where each regular language captures the projection of the transducer language on one of its tapes, and (3) we replace any occurrence of a variable x by a fresh copy of x that satisfies the same membership and length constraints as x. The resulting set of string constraints ψ' falls in the decidable fragment of the theory of strings with regular membership constraints and length constraints [2, 3]. Therefore, we can apply a similar technique as the one used in Norn [2, 3] to check the satisfiability of ψ'.

The rest of this section is organised as follows: First, we define the function Over that takes as input a constraint ϕ in ψ and transforms it into a set of constraints $\texttt{Over}(\phi)$ in ψ'. The form of $\texttt{Over}(\phi)$ will depend on the type of the constraint ϕ. Then, we formally define the set of constraints ψ' and show how to address its satisfiability problem. Finally, we show how to generate a new set of abstraction parameters that will be used by the under-approximation module in case that the set of constraints ψ' is satisfiable.

Transforming (dis-)equality constraints. Let us consider a constraint ϕ_i, with $i : 1 \le i \le k$, appearing in ψ. Let us assume that ϕ_i is an (dis-)equality constraint of the form $x_1 x_2 \cdots x_m \sim x_{m+1} x_{m+2} \cdots x_n$ with $\sim \in \{=, \neq\}$. Then $\texttt{Over}(\phi_i)$ will only contain the (dis-)equality constraint $(x_1, i, 1)(x_2, i, 2) \cdots (x_m, i, m) \sim (x_{m+1}, i, m +$

$1)(x_{m+2}, i, m+2)\cdots(x_n, i, n)$ where we replace any occurrence of a variable x by a fresh copy of the form (x, i, j).

Let $\texttt{Fresh}$ be a function that maps each variable x in $\mathbb{X}$ to its set of fresh copies. Formally, the set $\texttt{Fresh}(x)$ is the smallest set containing any variable of the form (x, i, j) such that ϕ_i is a (dis-)equality constraint of the form $x_1 x_2 \cdots x_m \sim x_{m+1} x_{m+2} \cdots x_n$, with $\sim \in \{\neq, =\}$, and $x_j = x$.

Transforming grammar constraints. In the following, we will show how to transform a grammar constraint ϕ in ψ into a set of regular constraints $\texttt{Over}(\phi)$ in ψ' using the function $\texttt{Over}$. This transformation is based on replacing the context-free language appearing in ϕ by a regular language that accepts its upward closure [4, 45], its downward closure [10, 44], or some other over-approximation, e.g., the one in [32]). To do that, we define a function $\texttt{Abst}$ that associates for each context-free grammar $\mathcal{G}$, a regular expression $\mathcal{R}$ such that $\mathcal{R}$ recognizes the upward/downward closure (or any other regular over-approximation) of the language generated by the context-free grammar $\mathcal{G}$.

Then, let us consider a grammar constraint ϕ_i of the form $x \in \mathcal{G}$. The set $\texttt{Over}(\phi_i)$ is then defined to be the smallest set containing all the regular constraints of the form $(x, \ell, j) \in \texttt{Abst}(\mathcal{G})$ where $(x, \ell, j) \in \texttt{Fresh}(x)$.

Transforming Regular Constraints. Let us consider a regular constraint ϕ_i of the form $x \in \mathcal{R}$. Then $\texttt{Over}(\phi_i)$ is defined as the smallest set containing all the regular constraints of the form $(x, \ell, j) \in \mathcal{R}$ where $(x, \ell, j) \in \texttt{Fresh}(x)$.

Transforming transducer constraints. In the following, we show how to replace a transducer constraint in ψ by a set of membership constraints in regular languages that capture an over-approximation of the transducer language. Each regular language captures the projection of the transducer language on one of its input tapes.

To compute these regular languages, we define a function $\texttt{Split}$ that takes as input a transducer $\mathcal{T}$ and outputs a pair of regular expressions $\mathcal{R}_1$ and $\mathcal{R}_2$ such that $\{(w_1, w_2) \,|\, w_2 \in \mathcal{T}(w_1)\} \subseteq (\llbracket \mathcal{R}_1 \rrbracket \times \llbracket \mathcal{R}_2 \rrbracket)$. Let us assume a transducer $\mathcal{T}$ of the form $\langle Q, \Sigma, \Delta, q_{init}, q_{acc} \rangle$. We can define then the automaton $\mathcal{A}_1 = \langle Q, \Sigma, \Delta_1, q_{init}, q_{acc} \rangle$ (resp. $\mathcal{A}_2 = \langle Q, \Sigma, \Delta_2, q_{init}, q_{acc} \rangle$) such that Δ_1 (resp. Δ_2) is the smallest transition relation containing $\langle q, a, q' \rangle \in \Delta_1$ (resp. $\langle q, b, q' \rangle \in \Delta_2$) if there is a transducer transition of the form $\langle q, \langle a, b \rangle, q' \rangle \in \Delta$. Let $\mathcal{R}_1$ (resp. $\mathcal{R}_2$) be the regular expression recognizing the same language as $\mathcal{A}_1$ (resp. $\mathcal{A}_2$). We then define $\texttt{Split}(\mathcal{T}) = (\mathcal{R}_1, \mathcal{R}_2)$.

Let us consider a transducer constraint ϕ_i of the form $y \in \mathcal{T}(x)$. The set $\texttt{Over}(\phi_i)$ is defined to be the smallest set containing all the regular constraints of the form $(x, \ell, j) \in \mathcal{R}_1$ where $(x, \ell, j) \in \texttt{Fresh}(x)$ and $(y, \ell', j') \in \mathcal{R}_2$ where $(y, \ell', j') \in \texttt{Fresh}(y)$.

Transforming length constraints. Let us consider a grammar constraint ϕ_i of the form $\sum_{1 \leq i \leq n} k_i \cdot \texttt{length}(x_i) \sim \ell$, where $x_i \in \mathbb{X}$, $k_i \in \mathbb{Z}$ for $i : 1 \leq i \leq n$, $\ell \in \mathbb{Z}$, and $\sim \in \{<, \leq, >, \geq, =\}$. Then, we define $\texttt{Over}(\phi_i)$ to be the smallest set of containing all constraints of the form $\sum_{1 \leq i \leq n} k_i \cdot \texttt{length}((x_i, \ell_i, j_i)) \sim \ell$ where $(x_i, \ell_i, j_i) \in \texttt{Fresh}(x_i)$.

Constructing the approximate set of constraints ψ'. In order to construct the set of constraints ψ', we need first to construct regular constraints that discard from the set of solutions any string that is accepted by any α-flat automaton with $\alpha \in \texttt{Covered}$. Let $\mathcal{R}_{\texttt{Covered}}$ be the regular expression that accepts the complement of the regular language $\bigcup_{\alpha \in \texttt{Covered}} \mathbb{F}^{\mathbb{X}}(\alpha)$. We use $\phi_{\texttt{Covered}}$ to denote the smallest set of constraints of the form $(x, \ell, j) \in \mathcal{R}_{\texttt{Covered}}$ where $(x, \ell, j) \in \texttt{Fresh}(x)$ for all variable $x \in \mathbb{X}$. We define ψ' as $\phi_{\texttt{Covered}} \cup \texttt{Over}(\phi_1) \cup \texttt{Over}(\phi_2) \cup \texttt{Over}(\phi_3) \cup \cdots \cup \texttt{Over}(\phi_k)$.

Satisfiability problem of the approximate set of constraints ψ'. The set of constraints ψ' satisfies the *acyclicity* condition defined in [2, 3]. Intuitively, the acyclicity condition is a syntactic condition on the occurrence of variables in the set of constraints and ensures that no variables appears more than once in (dis)-equalities during the analysis technique developed in [2, 3]. Thus, we can use the technique presented in [2, 3] to decide the satisfiability of the set of constraints ψ'. Then, let $\texttt{OAprx}(\texttt{Covered})$ be the algorithm that checks the satisfiability of ψ' and returns a satisfying assignment v for ψ' if ψ' is satisfiable, and $\texttt{unsat}$ otherwise.

Generating new set of abstraction parameters. In the following, we describe how to generate new set of abstraction parameters from an assignment v for ψ'. To do that, we will first show how to define the abstraction parameters for a string and then for an indexed string.

Let $w \in \Sigma^*$ be a string. We define $\texttt{GenPar}(w)$ to be the set of minimal pairs $\alpha = \langle \mathbb{p}, \mathbb{q} \rangle \in \mathbb{N}^2$ such that there are words $w_1, w_2, \ldots, w_{\mathbb{q}}$ where $\texttt{length}(w_i) \leq \mathbb{p}$ for $i : 1 \leq i \leq \mathbb{q}$ and $w \in w_1^* \bullet w_2^* \bullet \cdots \bullet w_{\mathbb{q}}^*$. Let $\mathbb{X}'$ be the set of variables appearing in ψ'. For an $\mathbb{X}'$-indexed string v over Σ, we define $\texttt{GenPar}(v)$ to be the maximal pairs in the set $\{\alpha \,|\, (x \in \mathbb{X}') \wedge (\alpha \in \texttt{GenPar}(v(x)))\}$.

10. CEGAR

In this section we present our CEGAR procedure. Observe that, due to the undecidability of the considered problem, our procedure is not guaranteed to terminate.

The procedure inputs a set ψ of string constraints. If the procedure terminates then it either returns an indexed string that satisfies ψ, or it concludes that ψ is not satisfiable.

The algorithm maintains a set $\texttt{Covered}$ of parameter values that have already been considered, and a set $\texttt{Waiting}$ of parameter values to be considered in the coming iterations. Both sets are initially empty. The procedure performs alternatively a sequence of over- and under-approximation phases.

The over-approximation phase is parameterized by the set $\texttt{Covered}$. There are two possible outcomes. If the over-approximation is unsatisfiable then we conclude that ψ is

Algorithm 5: CEGAR Procedure.

```
   Input: ψ: set of word constraints
   Output: ψ satisfiable?
1  Covered ← ∅;
2  Waiting ← ∅;
3  repeat
4      OAprxResult ← OAprx(Covered);
5      if OAprxResult = unsat then
6          return(unsat);
7      else if OAprxResult = v then
8          Waiting ← GenPar(v)
9      while Waiting ≠ ∅ do
10         Select and Remove α ∈ Waiting;
11         Covered ← Covered ∪ {α};
12         UAprxResult ← UAprx(α, ψ);
13         if UAprxResult = v then
14             return(v);
```

unsatisfiable and we terminate. Otherwise, we get a satisfying assignment θ. In such a case we use θ to generate a new set of parameters that we add to the set `Waiting`. This will ensure that we at least eliminate θ in the next iteration, possibly together with an infinite set of other valuations.

In the under-approximation phase, we check the elements of `Waiting` one by one, using the while-loop of line 9. Each time we select and remove a parameter α from `Waiting` and move to `Covered`. We check the under-approximation of ψ wrt. α. If the under-approximation produces a satisfying assignment then the procedure terminates.

11. Experimental Results

We have implemented our framework in an open source solver (called TRAU) using Z3 [11] as an SMT solver. We are not aware of other solvers that can handle the same set of string constraints without restricting the lengths of the solutions. Therefore, we have evaluated TRAU using two separate sets of benchmarks. First, we used the Kaluza benchmarks [38] in order to compare TRAU against existing state-of-the-art solvers for string equations with length and regular constraints but excluding context-free membership queries (CFG queries for short). Then, we used a set of string constraints with CFG queries in order to verify the absence of SQL injections. All experiments were performed on an Intel Core i7 2.7Ghz with 8GB RAM.

CFG-Free Benchmarks. The Kaluza suite [38] is an established set of benchmarks for string solvers. It was generated by a JavaScript symbolic execution engine. We use the SMT-format version provided by the CVC4 [26] team. The suite consists of approximately 50,000 queries, including length, regular and (dis)-equality constraints.

Figure 4a shows the performance of TRAU in comparison with three other state-of-the-art solvers: Z3-str2 [51], CVC4 [25, 26], and S3P [42]. The row "(*un*)*sat*" indicates the number of benchmarks for which the solvers decided sat/unsat. The row "*0-1s* (*5s, 10s, 20s*)" indicates the number of benchmarks for which the solvers are able to decide the outcome within the time limit of 1 (5, 10, 20) second(s). The row "*timeout*" indicates the number of benchmarks for which the solvers were unable to decide within the time limit of 20 seconds. Additionally, Z3-str2 detected "overlapping variables" on 525 examples and stopped running without giving a result. These cases are not shown in the figure. Due to the non-deterministic behavior of some of the other tools, they may exhibit slightly variable performances. We therefore carry out each experiment three times and consider the average result. As depicted in Figure 4a, TRAU can answer more queries than any of the three other tools. More importantly, it can handle hundreds of queries on which the other solvers timed out. These queries were typically the largest ones in terms of the number of string variables and the length of the discovered string solutions.

TRAU needs 5 iterations of CEGAR loop on average to handle a test in the test suite. The largest needed values of the abstraction parameter α is $\langle 7, 8\rangle$. Furthermore, when increasing the timeout limit to 100 seconds, TRAU is able to solve all the cases, including the ones for which a timeout is reported in the table.

An important hinder for the other solvers on these examples is their use of the *arrangement method* for solving word equations. For a string variable on a left-hand side of an equality, the arrangement method enumerates all possibilities of what sub-string of the right-hand side the variable could correspond to. Hence, the search space explored by the arrangement method is exponential in both the number of variables and the length of satisfying strings. Consequently, the running time grows quickly when the number of variables in the left-hand side and the length of the string in the right-hand side increase. Since the running time of our method is much less dependent on lengths of strings, it can handle these problematic cases much faster.

Benchmarks with CFG Queries. To our knowledge, all existing string solvers that allow CFG queries put a bound on the possible lengths of the string solutions. The HAMPI [24] solver can handle CFG queries but requires a priori bounding the length of the candidate string solutions. We have therefore generated our own set of benchmarks. The benchmarks use CFG queries in order to symbolically check for the possibility of SQL injections in a home made application.

Several web applications allow users to enter and save nested search queries. For instance, Bugzilla allows users to build Boolean combinations of simple facts about stored bug reports. Individual and group permissions are then typically used to control access to the entries on which the nested search queries are to be applied. SQL queries, such as `query = "SELECT * FROM records WHERE group=" + groupID + " AND " + userConjunction;` can then be

	CVC4	Z3-str2	S3P	TRAU
sat	33191	34459	34829	35202
unsat	11625	11747	12033	12019
0-1s	44562	32765	31321	38710
0-5s	44638	45922	44581	46502
0-10s	44703	46131	45846	47136
0-20s	44816	46249	46862	47221
timeout	2468	553	422	63

(a)

Input	Var	Length	TRAU				HAMPI	
			Bounded Length		Unbounded Length		Bounded Length	
			Result	Time(s)	Result	Times(s)	Result	Times(s)
cfg01	6	20	sat	1.14	sat	1.24	sat	0.52
cfg02	6	20	unsat	1.02	unsat	1.11	unsat	0.20
cfg03	8	50	sat	1.01	sat	1.45	sat	9.34
cfg04	8	50	unsat	1.56	unsat	1.54	unsat	9.33
cfg05	10	70	sat	1.55	sat	2.00	-	timeout
cfg06	10	70	unsat	2.01	unsat	1.12	-	timeout
cfg07	14	50	sat	2.13	sat	3.36	-	timeout
cfg08	14	50	unsat	1.56	unsat	2.58	unsat	8.85
cfg09	20	70	sat	1.78	sat	2.27	-	timeout
cfg10	20	70	unsat	2.46	unsat	1.89	-	timeout

(b)

Figure 4. (a) Performance of TRAU in comparison to CVC4, Z3-str2, and S3P on the Kaluza suite. (b) Performance of TRAU in comparison to HAMPI on the CFG suite.

used to return the entries that match the user supplied conjunction and her `groupID`. Without special care, an attacker can formulate nested conditions that allow her to bypass restrictions that apply to her `groupID`, for example by entering `(1 = 1) OR (1 = 1)` instead of a conjunction. Thus, sanitizers are used to parse and modify user inputs.

We have built such a sanitizer for nested SQL conditions. We use it to ensure that the entered conditions are conjunctions of (arbitrarily nested) SQL conditions. We then build SQL queries to submit to the underlying database. Following [41], we detect an SQL injection when the obtained query is a valid SQL query although the untrusted input (here the nested condition) is not derived from a single SQL-grammar-node (here a node for an arbitrary conjunction). Intuitively, in our case, an SQL injection occurs when the entered nested condition entered is not a conjunction (of arbitrarily nested conditions) yet yielding an overall valid query.

We have generated benchmarks for our solver by collecting the symbolic path conditions corresponding to walks through the sanitizer and requiring the obtained walks cannot be derived as and-conditions (the intended meaning of the input) when the whole query is a valid SQL condition. We have introduced "bugs" in our sanitizer in-order to allow for SQL injections, hence leading to satisfiable benchmarks. More specifically, we truncated some string terms without care for the succession of `'` symbols.

The results for some of the benchmarks are described in Figure 4b. The column *Var* gives the number of variables in the test. The column *Length* gives the bound on the length of string variables. Such a bound must be provided when running HAMPI. The column *Result* gives the answer of the solver: "(*un*)*sat*" means it is (im)possible to find values of the variables that satisfy the constraints. "-" denotes that the solver cannot finish the test. The column *Time* gives running time of the solver if it returns a result. Note that we supply two columns for TRAU: one where we fix an upper bound on the length of the possible solutions, and one where we do not. TRAU is the only solver we are aware of that can handle word equations with length constraints and CFG queries. We compare the performance of TRAU to HAMPI which has to bound the length of the solutions. Again, observe that TRAU is much less affected by the number of variables or by the length of the solutions. On the contrary, HAMPI is not efficient when the bounded length is larger than 50.

12. Concluding Remarks

We have presented a constraint solver for a rich language of constraints over unbounded strings, including word equations, context-free grammar membership, transducer constraints, and length constraints. The solver combines an under- and over-approximation scheme in a CEGAR loop, and is based on the observation that both satisfiability and unsatisfiability of common constraints can be demonstrated through witnesses with simple patterns. These patterns are captured using *flat* automata that consist of sequences of simple loops.

Acknowledgements

We are grateful to the reviewers of the paper and the tool. The depth of their feedback was very impressive, and it contributed to a substantial improvement both in the text and in the implementation.

This research has been partially supported by the Swedish Research Council, the Uppsala Programming for Multicore Architectures Research Center (UPMARC), CENIIT research organization (project 12.04), the MOST project no. 103-2221-E-001 -019 -MY3, the Czech Science Foundation project 16-24707Y, and the IT4IXS: IT4Innovations Excellence in Science project (LQ1602).

References

[1] P. A. Abdulla, B. Jonsson, M. Nilsson, and M. Saksena. A survey of regular model checking. In *CONCUR'04*, volume 170 of *LNCS*, pages 348–360. Springer, 2004.

[2] P. A. Abdulla, M. F. Atig, Y. Chen, L. Holík, A. Rezine, P. Rümmer, and J. Stenman. String constraints for verification.

In *CAV'14*, volume 8559 of *LNCS*, pages 150–166. Springer, 2014.

[3] P. A. Abdulla, M. F. Atig, Y. Chen, L. Holík, A. Rezine, P. Rümmer, and J. Stenman. Norn: An SMT solver for string constraints. In *CAV'15*, volume 9206 of *LNCS*, pages 462–469. Springer, 2015.

[4] M. F. Atig, A. Bouajjani, and T. Touili. On the reachability analysis of acyclic networks of pushdown systems. In *CONCUR'08*, volume 5201 of *LNCS*, pages 356–371. Springer, 2008.

[5] M. F. Atig, K. N. Kumar, and P. Saivasan. Acceleration in multi-pushdown systems. In *TACAS'16*, volume 9636 of *LNCS*, pages 698–714. Springer, 2016.

[6] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. CVC4. In *TACAS'11*, LNCS, pages 171–177. Springer, 2011.

[7] A. Brillout, D. Kroening, P. Rümmer, and T. Wahl. An interpolating sequent calculus for quantifier-free Presburger arithmetic. *Journal of Automated Reasoning*, 47:341–367, 2011.

[8] J. R. Büchi and S. Senger. Definability in the existential theory of concatenation and undecidable extensions of this theory. *Z. Math. Logik Grundlagen Math.*, 34(4), 1988.

[9] A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani. The MathSAT5 SMT solver. In *TACAS'13*, LNCS, pages 93–107. Springer, 2013.

[10] B. Courcelle. On constructing obstruction sets of words. *Bulletin of the EATCS*, (44):178–186, 1991.

[11] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS'08*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[12] B. Dutertre. Yices 2.2. In *CAV'14*, volume 8559 of *LNCS*, pages 737–744. Springer, July 2014.

[13] J. Esparza and P. Ganty. Complexity of pattern-based verification for multithreaded programs. In *POPL'11*, pages 499–510. ACM, 2011.

[14] J. Esparza, P. Ganty, S. Kiefer, and M. Luttenberger. Parikh's theorem: A simple and direct automaton construction. *Inf. Process. Lett.*, 111(12):614–619, 2011.

[15] J. Esparza, P. Ganty, and R. Majumdar. A perfect model for bounded verification. In *LICS'12*, pages 285–294. IEEE Computer Society, 2012.

[16] J. Esparza, P. Ganty, and T. Poch. Pattern-based verification for multithreaded programs. *ACM Trans. Program. Lang. Syst.*, 36(3):9:1–9:29, 2014.

[17] V. Ganesh and M. Berzish. Undecidability of a theory of strings, linear arithmetic over length, and string-number conversion. *CoRR*, abs/1605.09442, 2016.

[18] V. Ganesh, M. Minnes, A. Solar-Lezama, and M. Rinard. Word equations with length constraints: What's decidable? In A. Biere, A. Nahir, and T. Vos, editors, *Hardware and Software: Verification and Testing*, volume 7857 of *LNCS*, pages 209–226. 2013.

[19] P. Ganty, R. Majumdar, and B. Monmege. Bounded underapproximations. *Formal Methods in System Design*, 40(2):206–231, 2012.

[20] S. Ginsburg. *The Mathematical Theory of Context-Free Languages*. McGraw-Hill, Inc., 1966.

[21] S. Ginsburg and E. H. Spanier. Bounded algol-like languages. *Transactions of the American Mathematical Society*, 113(2):333–368, 1964.

[22] P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based whitebox fuzzing. *SIGPLAN Not.*, 43(6):206–215, June 2008.

[23] S. Kausler and E. Sherman. Evaluation of string constraint solvers in the context of symbolic execution. In *ASE '14*, pages 259–270. ACM, 2014.

[24] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst. HAMPI: A Solver for String Constraints. In *ISSTA'09*, pages 105–116. ACM, 2009.

[25] T. Liang, A. Reynolds, C. Tinelli, C. Barrett, and M. Deters. A DPLL(T) theory solver for a theory of strings and regular expressions. In *CAV'14*, volume 8559 of *LNCS*, pages 646–662. Springer, 2014.

[26] T. Liang, A. Reynolds, C. Tinelli, C. Barrett, and M. Deters. CVC4, 2016. URL `http://cvc4.cs.nyu.edu/papers/CAV2014-strings/`.

[27] A. W. Lin and P. Barceló. String solving with word equations and transducers: towards a logic for analysing mutation XSS. In *POPL'16*, pages 123–136. ACM, 2016.

[28] Z. Long, G. Calin, R. Majumdar, and R. Meyer. Language-theoretic abstraction refinement. In *FASE'12*, volume 7212 of *LNCS*, pages 362–376. Springer, 2012.

[29] R. Madhavan, M. Mayer, S. Gulwani, and V. Kuncak. Automating grammar comparison. *SIGPLAN Not.*, 50(10):183–200, Oct. 2015.

[30] G. Makanin. The problem of solvability of equations in a free semigroup. *Mathematics of the USSR-Sbornik*, 32(2):129–198, 1977.

[31] Y. Matiyasevich. Computation paradigms in light of hilberts tenth problem. In *New Computational Paradigms*, pages 59–85. Springer, New York, 2008.

[32] M. Mohri and M.-J. Nederhof. *Regular Approximation of Context-Free Grammars through Transformation*, pages 153–163. Springer Netherlands, Dordrecht, 2001.

[33] R. Parikh. On context-free languages. *J. ACM*, 13(4), 1966.

[34] W. Plandowski. Satisfiability of word equations with constants is in PSPACE. In *FOCS*, pages 495–500, 1999.

[35] W. Plandowski. An efficient algorithm for solving word equations. In *STOC*, pages 467–476, 2006.

[36] W. V. Quine. Concatenation as a basis for arithmetic. *The Journal of Symbolic Logic*, 11(4):105–114, 1946.

[37] J. M. Robson and V. Diekert. On quadratic word equations. In *STACS*, pages 217–226, 1999.

[38] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A Symbolic Execution Framework for JavaScript. In *IEEE Symposium on Security and Privacy*, pages 513–528. IEEE Computer Society, 2010.

[39] P. Saxena, S. Hanna, P. Poosankam, and D. Song. FLAX: Systematic discovery of client-side validation vulnerabilities in rich web applications. In *NDSS*. The Internet Society, 2010.

[40] K. U. Schulz. Makanin's algorithm for word equations - two improvements and a generalization. In *IWWERT*, pages 85–150, 1990.

[41] Z. Su and G. Wassermann. The essence of command injection attacks in web applications. *SIGPLAN Not.*, 41(1):372–382, Jan. 2006.

[42] M. Trinh, D. Chu, and J. Jaffar. Progressive reasoning over recursively-defined strings. In *CAV'16*, volume 9779 of *LNCS*, pages 218–240. Springer, 2016.

[43] M.-T. Trinh, D.-H. Chu, and J. Jaffar. S3: A symbolic string solver for vulnerability detection in web applications. In *CCS'14*, pages 1232–1243. ACM, 2014.

[44] J. van Leeuwen. A generalisation of parikhs theorem in formal language theory. In *ICALP*, volume 14 of *LNCS*, pages 17–26, 1974.

[45] J. van Leeuwen. Effective constructions in well-partially-ordered free monoids. *Discrete Mathematics*, 21(3):237 – 252, 1978.

[46] H. Wang, T. Tsai, C. Lin, F. Yu, and J. R. Jiang. String analysis via automata manipulation with logic circuit representation. In *CAV'16*, volume 9779 of *LNCS*, pages 241–260. Springer, 2016.

[47] G. Wassermann and Z. Su. Sound and precise analysis of web applications for injection vulnerabilities. *SIGPLAN Not.*, 42 (6):32–41, June 2007.

[48] F. Yu, M. Alkhalaf, and T. Bultan. Stranger: An automata-based string analysis tool for PHP. In *TACAS'10*, volume 6015 of *LNCS*, pages 154–157. Springer, 2010.

[49] Y. Zheng, X. Zhang, and V. Ganesh. Z3-str: a z3-based string solver for web application analysis. In *ESEC/FSE'13*, pages 114–124. ACM, 2013.

[50] Y. Zheng, X. Zhang, and V. Ganesh. Z3-str: A Z3-based string solver for web application analysis. In *ESEC/FSE'13*, pages 114–124. ACM, 2013.

[51] Y. Zheng, V. Ganesh, S. Subramanian, O. Tripp, J. Dolby, and X. Zhang. Effective search-space pruning for solvers of string equations, regular expressions and length constraints. In *CAV'15*, volume 9206 of *LNCS*, pages 235–254. Springer, 2015.