Flatten and Conquer
(A Framework for Efficient Analysis of String Constraints)

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 (CE-GAR) 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.

1. Introduction
Background. There has been a substantial amount of research in recent years on the development of solvers for string constraints [3, 21, 27, 32, 37]. 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 type of web vulnerabilities, may be used by attackers to bypass access controls and is typically caused by improper handling of strings by web applications [23]. 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 undecidable [7]. 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 that the diversity of the application areas 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 applications [30, 35], for comparing grammars or reasoning about the ambiguities or correctness of parsers [25], or for enabling deeper symbolic testing [18]. 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 [30]). In addition, precisely tracking all possible queries requires the ability to precisely 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 parameters consisting of a pair \(\alpha = (p, q)\) of natural numbers. A run of a flat automaton iterates a sequence of \(q\) loops each corresponding to a fixed word of length at most \(p\) (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.

Our framework is applicable to any class of constraints satisfying a sufficient condition which states that checking the emptiness of the flattening of any constraint can be effectively reduced to the satisfiability of a quantifier-free Presburger formula. We show that our sufficient condition is satisfied by a wide class of constraints. For instance, for a constraint \(\phi\) that requires membership in a context-free grammar \(G\), we show that we can derive a new grammar \(G'\) that captures the flattening of \(\phi\) (i.e., the set of strings that can be accepted by \(G\) and by the flat automaton). Then our sufficient condition follows since Parikh images of context-free languages are computable in general, and hence we can in particular compute the Parikh image of \(G'\). Further-
more, we show that the flattening of a word equation can be captured using a finite-state automaton. This implies our sufficient condition by computability of Parikh images for 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. Furthermore, 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 [10], CVC4 [6], Princess [8], MathSat [9], or Yices [11].

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 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.

**Overview.** Our procedure for solving string constraints is depicted in Fig. 1. The procedure inputs a set \( \psi \) of string constraints. If it terminates then it either returns the value \( \bot \) which means that \( \psi \) is unsatisfiable, or it returns a solution \( v \) to \( \psi \). In general, termination is not guaranteed. Recall that the problem of solving string constraints is undecidable. The procedure consists of a sequence of under- and over-approximation phases, one followed by the other. We maintain a set \( \text{Waiting} \) of abstraction parameters. Each iteration of the under-approximation module selects and removes one such a parameter \( \alpha \) from the set. The parameter \( \alpha \) is moved to the set \( \text{Covered} \) that contains all the abstraction parameters that have already been considered by the under-approximation module. The procedure flattens the input set \( \psi \) wrt. \( \alpha \), and computes its Parikh image as a quantifier-free Presburger formula \( \rho \) which is given to the SMT solver. If \( \rho \) is satisfiable, then the SMT solver will output a satisfying assignment that is translated to a satisfying assignment \( v \) of \( \psi \). 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 \( \text{Waiting} \) and repeats the loop. If the set is empty, then we have run out of parameters. This ends the current under-approximation phase, and triggers an over-approximation phase. The over-approximation procedure can use the set \( \text{Covered} \) 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 \( \text{Covered} \). Therefore, we know that \( \psi \) 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 \( \psi \) 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 \( \psi \). 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. More precisely, we generate the minimal elements of the set of all abstraction parameters whose corresponding automata accept \( v \), and put them in the set \( \text{Waiting} \). Our experiments indicate that these minimal elements have often small values even for large strings. Now, the over-approximation phase terminates and the next under-approximation phase starts. Notice that parameters that have been added to the set \( \text{Waiting} \) 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, then \( v \) 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 \( \psi \) 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 \( G \) by a membership constraint in a regular language that accepts the upward closure of the language of \( G \) [4, 33], (2) replacing a transducer constraint by a conjunction of membership constraints in regular languages that capture an over-approximation of the transducer language where each regular language captures the projection of the transducer language on one of its tapes, and (3) ensuring that any variable appears only once in the set of (dis-)equality constraints by replacing any occurrences of a variable \( x \) by fresh copies that satisfy the same membership and length constraints as \( x \). The resulting set of string constraints \( \psi' \) 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.

**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.
- 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

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 [20, 27, 28], 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 [37], CVC4 [21], S3 [32], 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 [23] 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 [36] soundly over-approximates string constraints using regular languages, and outperforms DPLL(T)-based solvers when checking single execution traces, according to some evaluations [19]. It has recently also been observed [34] 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 [17, 29]) have been also used in the context of verification of concurrent recursive programs (e.g., [5, 12, 14–16, 24]). In particular the work [24] 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 fundamentally (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. 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 \( \Sigma \) be a finite alphabet. We use \( \Sigma^* \) to denote the set of finite strings over \( \Sigma \), and use \( \epsilon \) to denote the empty string. We define \( \Sigma^* = \Sigma \cup \{\epsilon\} \).

For a string \( w \in \Sigma^* \), we use \( \text{length}(w) \) to denote the length of \( w \). A language \( L \) over \( \Sigma \) is a set \( L \subseteq \Sigma^* \). For strings \( w \) and \( w' \), we write \( w \leq w' \) to denote that \( w \) is a (not necessarily contiguous) substring of \( w' \). For a string \( w = a_1a_2\ldots a_n \in \Sigma^* \) and \( \Sigma' \subseteq \Sigma \), we define \( [w]_{\Sigma'} \) to be the largest (not necessarily contiguous) substring \( a_{i_1}a_{i_2}\ldots a_{i_m} \) of \( w \) such that \( a_{i_j} \in \Sigma' \), i.e., we remove from \( w \) the elements that are not members of \( \Sigma' \).

For a set \( X \), an \( X \)-indexed string over \( \Sigma \) is a mapping \( v : X \rightarrow \Sigma^* \), i.e., it assigns to each \( x \in X \), a string \( v(x) \) over \( \Sigma \). An \( X \)-indexed language \( K \) over \( \Sigma \) is a set of \( X \)-indexed strings over \( \Sigma \). For \( X \)-indexed languages \( K_1, K_2 \), we use \( K_1 \cap K_2 \) to denote their intersection, i.e., for each \( x \in X \), \( v \in (K_1 \cap K_2) \) iff both \( v \in K_1 \) and \( v \in K_2 \).

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

Automata and Grammars. A Finite-State Automaton (FSA) is a tuple \( A = (Q, \Sigma, \Delta, q_{\text{init}}, q_{\text{acc}}) \), where \( Q \) is a finite set of states, \( \Sigma \) is a finite alphabet, \( \Delta \subseteq Q \times \Sigma^* \times Q \) is a finite set of transitions, \( q_{\text{init}} \in Q \) is the initial state, and \( q_{\text{acc}} \in Q \) is the accepting state. A Regular Expression (RE) \( R \) over \( \Sigma \) is built inductively by including the empty expression \( \varepsilon \), the members of \( \Sigma \), and closing under the union \( + \), concatenation \( \cdot \), and Kleene star operator \( * \). A Context-Free Grammar (CFG) \( G = (N, T, P, S) \), 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 \( A \rightarrow a \), where \( A \in N \), and \( a \in (N \cup T)^+ \). We call \( a \) the rhs of \( p \). The languages \( [A], [R], [G] \) of \( A, R, G \) are defined in the standard manner.

A transducer \( T \) is of the same form as an FSA, the only difference being that now \( \Delta \subseteq Q \times X \times \Sigma \times Q \). For strings \( w_1, w_2 \in \Sigma^* \), we write \( w_2 \in T(w_1) \) to denote that there is a sequence \( q_0 = (a_1,b_1)q_1(a_2,b_2)\ldots(a_n,b_n)q_n \) such that \( q_0 = q_{\text{init}}, q_n = q_{\text{acc}}, (q_i(a_{i+1},b_{i+1}),q_{i+1}) \in \Delta \) for all \( i : 0 \leq i \leq n, w_1 = a_1a_2\ldots a_n \), and \( w_2 = b_1b_2\ldots b_n \).

Presburger Formulas. Presburger arithmetic is the first-order theory of the natural numbers with addition. Here, we introduce a subset of its formulas that we will simply refer to as Presburger formulas. Fix a finite set \( \mathcal{F} \) of numerical variables, i.e., variables ranging over \( \mathbb{N} \). A linear constraint \( \varphi \) over \( \mathcal{F} \) is of the form \( \exists y_1y_2\ldots y_m \sum_{1 \leq i \leq n} k_i \cdot x_i = k \) where \( k_i \in \mathbb{Z} \) for \( i : 1 \leq i \leq n, k \in \mathbb{Z}, \neg e \in \{<,\leq,\geq,=\} \), and \( \{x_1,x_2,\ldots,x_n\} \subseteq \{y_1,y_2,\ldots,y_m\} \subseteq \mathbb{N} \). For a valuation \( \theta : \mathcal{F} \rightarrow \mathbb{N} \), we write \( \theta = \varphi \) to denote that there are \( a_1,a_2,\ldots,a_m \in \mathbb{N} \) such that \( \sum_{1 \leq i \leq n} k_i \cdot \theta'(x_i) = k \), holds, where \( \theta'(x) = \theta(x) \) if \( x \in \{x_1,x_2,\ldots,x_n\} \) and \( \theta'(x) = a_j \) if \( x = y_j \) for some \( j : 1 \leq j \leq m \). Sometimes we use a set notation for the existential quantifiers, and write \( \varphi \) as \( \exists A. \sum_{1 \leq i \leq n} k_i \cdot x_i = k \) where \( A = \{y_1,y_2,\ldots,y_m\} \). A Presburger constraint \( \rho \) over \( \mathcal{F} \) consists of a set of atoms, each of which is a linear constraint over \( \mathcal{F} \). The atoms are combined through a number of conjunctions and disjunctions. For \( \theta : \mathcal{F} \rightarrow \mathbb{N} \), we write \( \theta = \rho \) to denote that \( \rho \) evaluates to true when the atoms are evaluated under \( \theta \) as described above, and the results are combined using the conjunctions and disjunctions in \( \rho \). We define \( [\rho] := \{\theta : \theta = \rho\} \). We assume a function \( \text{SMT} \) which, given a Presburger constraint \( \rho \), either \( \text{SMT}(\rho) = \theta \) for some \( \theta \) with \( \theta \in [\rho] \), or \( \text{SMT}(\rho) = \emptyset \).

Parikh Images. Consider an alphabet \( \Sigma \). For a string \( w \in \Sigma^* \), we define \#\( w \) : \( \Sigma \rightarrow \mathbb{N} \) to be a function such that, for each symbol \( a \in \Sigma \), \#\( w(a) \) gives the number of occurrences of \( a \) in \( w \). The Parikh image of a language \( L \subseteq \Sigma^* \) is defined by \#\( L \) := \{\#\( w \) : w \in L\} \). We will characterize the Parikh image of some languages using Presburger formulas. To do that, we define the set \( \Sigma^* = \{a^* \mid a \in \Sigma\} \), where \( a^* \) 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 Presburger constraint over \( \Sigma^* \), denoted \( \text{CompP}(L) \) (for Compute Parikh image), that characterizes the Parikh image of \( L \). More precisely, for any \( \theta : \Sigma^* \rightarrow \mathbb{N} \), we have \( \theta = \text{CompP}(L) \) if there is a string \( w \in L \) such that \( \theta(a^*) = \#\( w(a) \) 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 \( G \), we can compute \( \text{CompP}(G) \) in polynomial time [13, 26]. Notice that this implies that we can also compute \( \text{CompP}(\{w \mid \exists w \} \) for a regular expression \( \mathcal{R} \) in polynomial time. For simplicity, we sometime write \( \text{CompP}(G) \) and \( \text{CompP}(\mathcal{R}) \) instead of \( \text{CompP}(\{w \mid \exists w \} \) and \( \text{CompP}(\mathcal{R}) \).

We extend the notion of Parikh images to indexed languages as follows. For an indexed string \( v : X \rightarrow \Sigma^* \), we define the mapping \#\( v \) : \( X \rightarrow \mathbb{N} \) such that, for each variable \( x \in X \) and symbol \( a \in \Sigma \), \#\( v(x)(a) \) gives the number of occurrences of \( a \) in \( v(x) \). The Parikh image of an \( X \)-indexed

![Figure 2: A (3, 2)-flat automaton of \((ab)^* \bullet (baa)^*\).](image-url)
language $K$ over $\Sigma$ is defined by $\#K := \{ \#v \mid v \in K \}$. We consider the set \( (\times \times \Sigma)^* := \{ (x, a)^* \mid (x \in \times) \land (a \in \Sigma) \} \), and say that $K$ is Parikh-definable if there is a Presburger constraint over $(\times \times \Sigma)^*$ such that, for any $\theta : (\times \times \Sigma)^* \rightarrow \mathbb{N}$, we have $\theta = \text{CompP}(K)$ iff there is an $\times$-indexed string $w \in K$ such that $\theta((x, a)^*) = \#v(x)(a)$ for all variables $x \in \times$ and symbols $a \in \Sigma$.

4. String Constraints

Fix a finite alphabet $\Sigma$ and a finite set of variables $\times$ ranging over $\Sigma^*$. Below, we define a set of string constraints over $\Sigma$ and $\times$. Each constraint $\phi$ characterizes an $\times$-indexed language $[\phi]$ over $\Sigma$. The set of terms $\text{Terms}(\Sigma, \times)$ over $\Sigma$ and $\times$ is the smallest set such that (i) $\text{Terms}(\Sigma, \times) \subseteq \text{Terms}(\Sigma, \times)$, and (ii) if $t_1, t_2 \in \text{Terms}(\Sigma, \times)$ then $t_1 \bullet t_2 \in \text{Terms}(\Sigma, \times)$. Given an $\times$-indexed string $v : \times \rightarrow \Sigma^*$ over $\Sigma$, we extend it to terms by defining $v : \text{Terms}(\Sigma, \times) \rightarrow \Sigma^*$. We define $[\phi] := \{ v \mid v \in \times \rightarrow \Sigma^* \}$. An equality constraint is of the form $t = t_2$ where $t_1, t_2 \in \text{Terms}(\Sigma, \times)$. We define $[\phi] := \{ v \mid v(t_1) = v(t_2) \}$. A disequality constraint is of the form $t_1 \neq t_2$ and is interpreted analogously.

A transducer constraint $\phi$ is of the form $x \in T(x)$ where $x, y \in \times$ and $T$ is a transducer. We define $[\phi] := \{ v \mid v(y) \in T(v(x)) \}$. A grammar constraint $\phi$ is of the form $x \in G$, where $x \in \times$ and $G = (N, T, P, S)$ is a CFG with $T = \Sigma$. We define $[\phi] := \{ v \mid v(x) \in \llbracket G \rrbracket \}$. The special case of regular constraints, of the form $x \in R$ where $R$ is a regular expression over $\Sigma$ is interpreted in a similar manner.

A length constraint $\phi$ is of the form $\sum_{x \in \times} k_i \cdot \text{length}(x_i) = k$, where $\sim \in \prec, \succ, \; = \}$, $x_i \in \times$, and $k_i, k \in \mathbb{Z}$. We define $[\phi] := \{ v \mid \sum_{x \in \times} k_i \cdot \text{length}(v(x_i)) = k \}$. A string constraint is of one of the above forms. A set $\psi$ of constraints is interpreted as $[\psi] := \cap_{\phi \in \psi} [\phi]$.

5. Flat Languages

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

5.1 Flat Languages

A language $L$ over $\Sigma$ is said to be $\alpha$-flat if there are strings $w_1, w_2, \ldots, w_n \in \Sigma^*$ such that $\text{length}(w_i) \leq p$, for each $i : 1 \leq i \leq q$, and $L = (w_1)^* \bullet (w_2)^* \bullet \cdots \bullet (w_n)^*$. We call $w_1, w_2, \ldots, w_n$ the loops of $L$. We can recognize an $\alpha$-flat language over $\Sigma$ using a special class of automata, which we call $\alpha$-flat automata. A $(3, 2)$-flat automaton is shown in Fig. 2. The automaton recognizes the $(3, 2)$-flat language $(ab)^* \bullet (baa)^*$. The automaton contains two loops (cycles), each with three states. Below, we define formally the notion of an $\alpha$-flat automaton. We define $\mathcal{S}(\alpha) := \mathbb{p} \cdot \mathbb{q}$, giving the number of states in the automaton ($\alpha$ loops each with $p$ states), and define $\mathcal{S}(\alpha) := \{ i \mid 1 \leq i \leq \mathcal{S}(\alpha) \}$, i.e., we enumerate the states from 1 to $\mathcal{S}(\alpha)$. We define the set $\text{Entries}(\alpha) := \{ i \mid (i \in \mathcal{S}(\alpha)) \land (i \text{ mod } \mathbb{p} = 1) \}$ which gives the states that are entries of loops in the automaton; $\text{LastEntry}(\alpha) := \mathcal{S}(\alpha) - \{ \mathbb{p} \cdot 1 \}$, 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 $\text{Loop Succ}(\alpha)(i) := \{ i + 1 \}$ if $i \text{ mod } \mathbb{p} \neq 0$, and $\text{Loop Succ}(\alpha)(i) := \{ i - \mathbb{p} + 1 \}$ if $i \text{ mod } \mathbb{p} = 0$, i.e., the function gives the (single) successor of the state that lies within the same loop. We define $\text{Entry Succ}(\alpha)(i) := \{ i + \mathbb{p} \}$ if $i \in \text{Entries}(\alpha) - \{ \text{LastEntry}(\alpha) \}$, and $\text{Entry Succ}(\alpha)(i) := \emptyset$ otherwise, i.e., for a loop entry, the function gives the next loop entry. We define $\text{succ}^*(\alpha)(i) := \text{Loop Succ}(\alpha)(i) \cup \text{Entry Succ}(\alpha)(i)$. 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 $\text{succ}^*(\alpha)(i)$ to denote the reflexive transitive closure of $\text{succ}(\alpha)(i)$.

For example, we have $\mathcal{S}(3, 2) = 6$, $\text{Entries}(3, 2) = \{ 1, 4 \}$, $\text{Last Entry}(3, 2) = 4$, $\text{Loop Succ}(3, 2)(1) = \{ 2 \}$, $\text{Entry Succ}(3, 2)(1) = \{ 4 \}$, $\text{succ}(3, 2)(1) = \{ 2, 4 \}$, $\text{succ}(3, 2)(2) = \{ 3 \}$, and $\text{succ}^*(3, 2)(4) = \{ 4, 5, 6 \}$. Formally, an $\alpha$-automaton $A$ is a tuple $(Q, \Sigma, \Delta, q_{\text{init}}, q_{\text{acc}})$, where (i) $Q = \mathcal{S}(\alpha)$, (ii) $\Delta = \Delta' \cup \Delta''$. The set $\Delta'$ contains for each $i, j$ with $j \in \text{Loop Succ}(\alpha)(i)$, (one and only one) transition of the form $(i, a, j)$ where $a \in \Sigma$. The set $\Delta''$ contains for each $i, j$ with $j \in \text{Entry Succ}(\alpha)(i)$, (one and only one) transition of the form $(i, e, j)$, i.e., transitions between two consecutive loop entries are always labeled with $e$. (iii) $q_{\text{init}} = 1$, and (iv) $q_{\text{acc}} = \text{Last Entry}(\alpha)$, i.e., the accepting state is the entry of the last loop. Notice that, for a given parameter $\alpha$, all $\alpha$-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 $e$-transitions, we essentially allow loops of sizes up to $p$ (rather than equal to $p$), and allow up to $q$ loops (rather than exactly $q$ loops). Given $p$ and $q$, there are $\sum_{i=0}^{p-1} q^i$ different $\alpha$-flat automata over $\Sigma$ (since each such an automaton contains $p \cdot q$ transitions inside its
loops, each of which may be labeled by some element in $\Sigma$). We define the complete $\alpha$-flat language over $\Sigma$, by $G(\alpha) := \bigcup \{ L \mid L \text{ is an $\alpha$-flat language over } \Sigma \}$, i.e., it is the union of all $\alpha$-flat languages over $\Sigma$. For a set $X$ of variables, we define the complete $X$-indexed $\alpha$-flat language over $\Sigma$ by $G^X(\alpha) := \{ x : X \Rightarrow \Sigma^* \mid \forall x \in X, v(x) \in G(\alpha) \}$, i.e., it is the set of $X$-indexed strings over $\Sigma$ such that each variable is mapped to a string in $G(\alpha)$.

5.2 Generic Flat Automata

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

We will consider the alphabet $\Sigma(a) := \{ a(i) \mid (a \in \Sigma) \wedge (i \in S(a)) \} \cup \{ \varepsilon \}$. We define the generic $\alpha$-flat automaton over $\Sigma, B(\alpha) := \{ Q, \Sigma(a), \Delta, q_{init}, q_{acc} \}$ (Fig. 3), where $Q, q_{init},$ and $q_{acc}$ are of the same form as for $\alpha$-flat automata, and $\Delta = \Delta' \cup \Delta''$ with $\Delta' = \{ (i, a(i), j) \mid (a \in \Sigma) \wedge (j \in LoopSucc(a(i))) \}$, and $\Delta'' = \{ (l, i, j) \mid j \in EntrySucc(a(i)) \}$. 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 $\epsilon$-transitions between the consecutive loop entries. We define the $\alpha$-generic language $G(\alpha) := [B(\alpha)], i.e., it is the language of $B(\alpha)$. A string over $\Sigma$ is said to be $\alpha$-generic (or simply generic) if it belongs to $G(\alpha)$. The generic $\alpha$-flat automaton encodes the behaviors of all $\alpha$-automata. More precisely, given an $\alpha$-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 $X$-indexed language $G^X(\alpha) := \{ x : X \Rightarrow \Sigma^* \mid \forall x \in X, v(x) \in G(\alpha) \}$. An $X$-indexed string $v$ is $\alpha$-generic if it belongs to $G^X(\alpha)$.

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 $i \in S(\alpha)$ 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 $v \in X \Rightarrow \Sigma^*$ is said to be pure if $v$ is pure for all $x \in X$. We define the language $F^X(\alpha) := \{ x : X \Rightarrow \Sigma^* \mid v \text{ is pure} \}$. An $X$-indexed language $K$ over $\Sigma$ is said to be $\alpha$-generic if $K \subseteq F^X(\alpha)$, and it is called pure if $K \subseteq G^X(\alpha)$.

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 $GetS$ such that for any $\theta : (X \times \Sigma(\alpha))^* \Rightarrow \mathbb{N}$, the value of $GetS(\theta)$ is the unique $X$-string $v \in G^X(\alpha) \cap F^X(\alpha)$ with $\theta(x, a(i)) = \#v(x)(a(i))$ for all variables $x \in X$ and symbols $a \in \Sigma$. Notice that $GetS(\theta)$ may not exist. However, if it exists then, by Lemma 1, it is unique. We get the following Corollary.

Corollary 1. For an $X$-indexed language $K \subseteq (G^X(\alpha) \cap F^X(\alpha))$, if $\theta \in \#K$ then $GetS(\theta) \in K$.

Also, Lemma 1 implies the following lemma.

Lemma 2. For $X$-indexed languages $K_1, K_2 \subseteq (G^X(\alpha) \cap F^X(\alpha))$, 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.

6. Flattening

Fix a set of variables $X$, an alphabet $\Sigma$, parameters $p, q \in \mathbb{N}$, and $\alpha = \langle p, q \rangle$. We will describe how to construct the flattening of a string constraint $\phi$ in different forms as defined in Section 4, which corresponds to taking the intersection of $[\phi]$ with the generic $\alpha$-flat automaton. We will take the flattening of $\phi$ and intersect it with the set of pure languages thus obtaining a particular $X$-indexed language $[\phi]_{\alpha}$ that satisfies two important properties. First, $[\phi]_{\alpha}$ characterizes the intersection of $\phi$ and flat languages in the sense that any indexed string in $[\phi]_{\alpha}$ can be renamed to an indexed string that is in the intersection of $[\phi]$ and flat languages. (ii) $[\phi]_{\alpha}$ is Parikh definable (see Section 7.)

6.1 Flattening Grammar Constraints

Consider a grammar constraint $\phi$ of the form $x \in G$ with $G = \langle N, T, P, S \rangle$, terminalset $= \Sigma$, and a parameter $\alpha = \langle p, q \rangle$. We will define a new grammar Flatten($\alpha$)($\phi$) which encodes running $G$ “in parallel” with the $\alpha$-generic automaton. We define 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 $i, j \in S(\alpha)$ with $j \neq \text{succ}^*(\alpha)(i)$, the set $N_1'$ contains a corresponding nonterminal $A^\theta(i, j)$, i.e., $N_1' := \{ A^\theta(i, j) \mid (A \in N) \land (j \neq \text{succ}^*(\alpha)(i)) \}$.

Intuitively, the next segment of the input string expected by $G$ corresponds to $A$, while the automaton is currently in state $i$. We use $A^\theta$ 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 $j$.  

6
• For each \( a \in T \) and \( i, j \in \mathcal{S}(a) \) with \( j \in \text{succ}^+ (a) (i) \), the set \( N_2^j \) contains a corresponding nonterminal \( a^\oplus (i, j) \), i.e., \( N_2^j = \{ a^\oplus (i, j) \mid (a \in T) \land (j \in \text{succ}^+ (a) (i)) \} \). Intuitively, the next terminal expected by \( \mathcal{G} \) is \( a \). The automaton is currently in the state \( i \), may perform an arbitrary number of \( \epsilon \)-transitions both before and after performing a transition labeled with \( a \), and ends up in the state \( j \).

• For each \( i, j \in \mathcal{S}(a) \) with \( j \in \text{succ}^+ (a) (i) \), the set \( N_3^j \) contains a corresponding nonterminal \( e^\oplus (i, j) \), i.e., \( N_3^j = \{ e^\oplus (i, j) \mid j \in \text{succ}^+ (a) (i) \} \). This allows the automaton to perform an arbitrary number of \( \epsilon \)-transitions.

We define the start symbol \( S' = S(1, \text{LastEntry}(a)) \). We 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 \( A \rightarrow X_1 \cdot X_2 \cdot \ldots \cdot X_n \cdot x \), and \( i, j \in \mathcal{S}(a) \) with \( j \in \text{succ}^+ (a) (i) \), the set \( P_1^* \) contains all productions of the form \( A^\oplus (i, j) \rightarrow X_1^\oplus (i, i_1) \cdot X_2^\oplus (i_1, i_2) \cdot \ldots \cdot X_{n-1}^\oplus (i_{n-1}, i_n) \cdot x \), where \( i_0 = i, i_n = j, i_k \in \text{succ}^+ (a) (i_{k+1}) \), for \( k : 1 \leq k \leq n \). The next segment of the input string can be consumed by \( \mathcal{G} \) and the automaton runs parallel on each sub-segment according to the right-hand sides of \( p \), by letting \( \mathcal{G} \) and the automaton run parallel on each sub-segment.

• For each terminal \( a \in T \) and \( i, j \in \mathcal{S}(a) \) with \( j \in \text{succ}^+ (a) (i) \), the set \( P_2^* \) contains all productions of the form \( a^\oplus (i, j) \rightarrow e^\oplus (i_0, i_1) \cdot e^\oplus (i_1, i_2) \cdot \ldots \cdot e^\oplus (i_{n-1}, i_n) \cdot e^\oplus (i_n, j) \), where \( i_0 = i, i_n = j, i_k \in \text{succ}^+ (a) (i_{k+1}) \), and \( i_k \in \text{succ}^+ (a) (i_k) \). The automaton is allowed to perform an arbitrary number of \( \epsilon \)-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 \( \epsilon \)-transitions)

  • All productions that are of the form \( e^\oplus (i, j) \rightarrow e^\oplus (k, j) \cdot e^\oplus (k, j) \), where \( i, j, k \in \mathcal{S}(a), k \in \text{LoopSucc} (a) (i) \), and \( j \in \text{succ}^+ (a) (k) \), i.e., the automaton performs one \( \epsilon \)-transition to the state \( i \) and then takes some number of \( \epsilon \)-transitions to the state \( j \).

  • All productions that are of the form \( e^\oplus (i, i) \rightarrow e^\oplus (i, i) \), where \( i \in \mathcal{S}(a) \), i.e., stopping \( \epsilon \)-generating transitions.

  • For all \( i, j \in \mathcal{S}(a) \) with \( j \in \text{EntrySucc} (a) (i) \), the set \( P_4^* \) contains the production \( A(i, j) \rightarrow e \). The automaton is allowed to cross from one loop entry to the next.

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

### 6.2 Flattening Equality Constraints

We consider an equality constraint \( \phi \) of the form \( x_1 \cdot x_2 \cdot \ldots \cdot x_m = x_{m+1} \cdot x_{m+2} \cdot \ldots \cdot x_n \), and a parameter \( a = \{ y, t \} \). A constant \( e \) in an equality constraint can be replaced by a fresh variable \( x \) with a regular constraint \( x \in [c] \). We will define an FSA \( \text{Flatten} (a) (\phi) \) 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 \( \Sigma (n, a) := \{ a(k, i) \mid (a \in \Sigma) \land (1 \leq k \leq n) \land (i \in \mathcal{S}(\alpha)) \} \). We define \( \text{Flatten} (a) (\phi) := (Q, \Sigma (n, a), \Delta, q_{\text{inst}}, q_{\text{acc}}) \) 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 \( i_1, i_2 \in \mathcal{S}(a) \), the set \( Q_1 \) contains the state \( (k, i_1, i_2) \). 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 \( \ell \) 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 \( k : 1 \leq k \leq m, \ell : m + 1 \leq \ell \leq n, i_1, i_2 \in \mathcal{S}(a), \) and \( a \in \Sigma \), the set \( Q_2 \) contains the set \( \{ k, i_1, i_2, a \} \). This state encodes that the automaton of \( x_k \) has just performed a transition labeled by \( a \). The automaton of \( x_{\ell} \) 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 \( a \in \Sigma, k, i_1, i_2, j \) with \( 1 \leq k \leq m, m + 1 \leq \ell \leq n, i_1, i_2, j \in \mathcal{S}(a), \) and \( j \in \text{LoopSucc} (a) (i_1) \), the set \( \Delta_1 \) contains the transition \( \{ (k, i_1, i_2, a) \} \). This corresponds to the case where the automaton of \( x_k \) performs a transition labeled with \( a \).

• For each \( a \in \Sigma, k, i_1, i_2, j \) with \( 1 \leq k \leq m, m + 1 \leq \ell \leq n, i_1, i_2, j \in \mathcal{S}(a), \) and \( j \in \text{LoopSucc} (a) (i_2) \), the set \( \Delta_2 \) contains the transition \( \{ (k, i_1, i_2, a) \} \). This corresponds to the case where the automaton of \( x_k \) performs a transition labeled with \( a \) (answering the previous move of the automaton of \( x_k \)).

• For each \( k, i_1, i_2, j \) with \( 1 \leq k \leq m, m + 1 \leq \ell \leq n, i_1, i_2, j \in \mathcal{S}(a), \) and \( j \in \text{LoopSucc} (a) (i_1) \), the set \( \Delta_3 \) contains the transition \( \{ (k, i_1, i_2, a) \} \). This corresponds to the case where the automaton of \( x_k \) makes an \( \epsilon \)-transition, while the automaton of \( x_{\ell} \) does not move.

• For each \( k, i_1, i_2, j \) with \( 1 \leq k \leq m, m + 1 \leq \ell \leq n, i_1, i_2, j \in \mathcal{S}(a), \) and \( j \in \text{LoopSucc} (a) (i_2) \), the set \( \Delta_4 \) contains the transition
\{(k, i_1, \ell, i_2), \epsilon(\ell, i_2), (k, i_1, \ell, j)\}. This case is symmetric to the previous one.

- For each \(k, i_1, \ell, i_2, j\) with \(1 \leq k \leq m, m + 1 \leq \ell \leq n, i_1, i_2, j \in S(\alpha)\), and \(j \in \text{EntrySucc}(\alpha)(i_1)\), the set \(\Delta_0\) contains the transition \(\{(k, i_1, \ell, i_2), \epsilon, (k, i_1, \ell, j)\}\). This corresponds to the case where the automaton of \(x_k\) crosses from one loop entry to the next.

- For each \(k, i_1, \ell, i_2, j\) with \(1 \leq k \leq m, m + 1 \leq \ell \leq n, i_1, i_2, j \in S(\alpha)\), and \(j \in \text{EntrySucc}(\alpha)(i_2)\), the set \(\Delta_0\) contains the transition \(\{(k, i_1, \ell, i_2), \epsilon, (k+1, j, \ell, i_2)\}\). This case is symmetric to the previous one.

- For each \(k, i_1, \ell, i_2, j\) with \(1 \leq k \leq m, m + 1 \leq \ell \leq n, i_1, i_2, j \in S(\alpha)\), and \(i_2 = \text{LastEntry}(\alpha)\), the set \(\Delta_0\) contains the transition \(\{(k, i_1, \ell, i_2), \epsilon, (k, i_1, \ell + 1, j)\}\). This case is symmetric to the previous one.

We define the initial state as \(q_{\text{init}} := (1, 1, 1, 1)\), 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 \(q_{\text{acc}} := (m, \text{LastEntry}(\alpha), n, \text{LastEntry}(\alpha))\), 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 \(\langle \phi \rangle_\alpha\) we need to give some definitions. First we formulate some conditions on the strings generated by \(\text{Flatten}(\alpha)(\phi)\). A string \(w \in (S(n, \alpha))^*\) is said to be rational if \(#a(k, i) \neq #a(l, i)\) whenever \(x_k = x_l\). In other words, different occurrences of the same variable will run the corresponding generic automaton in a similar 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 \(\#w(k, i) > 0\) implies \(\#w(b(k, i)) = 0\). We will use the purity of strings in \(\text{Flatten}(\alpha)(\phi)\) to guarantee the purity of \(\langle \phi \rangle_\alpha\). Next, we take the strings generated by \(\text{Flatten}(\alpha)(\phi)\) and project them on the variables that occur in \(\phi\). For a \(k : 1 \leq k \leq n\), we define the alphabet \(\Sigma_k := \{a(k, i) \mid (a \in \Sigma) \wedge (1 \leq i \leq S(\alpha))\}\). The set \(\Sigma(n, \alpha)\) contains 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 \(\langle \phi \rangle_\alpha\) contains all \(X\)-indexed strings \(v : X \mapsto \Sigma(\alpha)\) such that there is a string \(w \in \text{Flatten}(\alpha)(\phi)\) satisfying the following properties: (i) \(w\) is rational and pure. (ii) \(v(x) = \mathcal{R}(\{w[i]x\})\) 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 \(\mathcal{R}(\{w[i]x\})\) is \(\alpha\)-generic. (iii) \(v(x) \in C(\alpha) \cap \mathbb{F}(\alpha)\) if \(x \in \mathbb{X} - \{x_1, \ldots, x_n\}\). Such a variable is not restricted by \(\phi\) and hence it may be assigned any pure string in the generic \(\alpha\)-flat automaton. Notice that \(\langle \phi \rangle_\alpha\) is \(\alpha\)-generic.

### 6.3 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.

### 6.4 Properties

Lemma 3 and Lemma 4 below follow from the flattening construction. They explain the relation between \(\langle \phi \rangle_\alpha\) and the intersection of \(\phi\) with flat languages.

**Lemma 3.** \(\mathcal{R}^\alpha(\langle \phi \rangle_\alpha) \subseteq [\phi] \cap \mathbb{F}(\alpha)\).

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

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

**Theorem 1.** \(\langle \psi \rangle_\alpha = \emptyset\) iff \([\psi] \cap \mathbb{F}(\alpha) = \emptyset\).

From Corollary 1 and Lemma 3 we get the following.

**Theorem 2.** If \(\emptyset \in \#(\psi)_\alpha\) then \(\mathcal{R}^\alpha(\text{GetS}(\emptyset)) [\psi] \cap \mathbb{F}(\alpha) = \emptyset\).

### 7. Under-Approximation

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

### 7.1 Computing Parikh Images.

We give an algorithm for computing the Parikh image of \( \langle φ \rangle_α \cap F^x (α) \) 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 Presburger formulas over the alphabet \( (X × Σ(α))^* \). The algorithms are defined based on the construction of \( \langle φ \rangle_α \), described in Section 6. 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 \( x ∈ G \)). We compute the Parikh image of the automaton \( G \) wrt. the abstraction parameter \( α \) according to the construction of Section 6. We compute the Parikh image of \( G^t \) (this is possible since \( G^t \) is a CFG), and store the result in \( ρ_1 \). Notice that \( ρ_1 \) is defined over the set \( (Σ(α))^* \). We define the formula \( ρ_2 \) that renames each variable \( (a(i))^* \) to the corresponding variable \( (x, a(i))^* \). 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 \( ρ_3 \) is the conjunction of the previous three formulas. Furthermore, we quantify away all the variables in \( (Σ(α))^* \) thus ensuring that \( ρ_3 \) is defined over the alphabet \( (X × Σ(α))^* \).

**Equalities.** Algorithm 2 shows the case where \( φ \) is an equality constraint (of the form \( x_1 x_2 ... x_m = x_{m+1} x_{m+2} ... x_n \)). In a similar manner to the case of grammars, we compute the flattening \( G' \) of the grammar \( G \) wrt. the abstraction parameter \( α \) as described in Section 7.1. We give an algorithm for computing the Parikh image of \( G' \) (\( G' \) is a CFG), and store the result in \( ρ_1 \). Notice that \( ρ_1 \) is defined over the sets \( (Σ(α))^* \). We define the formula \( ρ_2 \) that renames each variable \( (a(i))^* \) to the corresponding variable \( (x, a(i))^* \). 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 \( ρ_3 \) is the conjunction of the previous three formulas. Furthermore, we quantify away all the variables in \( (Σ(α))^* \) thus ensuring that \( ρ_3 \) is defined over the alphabet \( (X × Σ(α))^* \).

### 7.2 SMT Solving

In Algorithm 3, we are given a set \( ψ \) of constraints together with an abstraction parameter \( α \). We construct, for each \( φ ∈ ψ \), the Parikh image of \( \langle φ \rangle_α \), as described in Section 7.1. We collect the conjunction of the Parikh images in \( ρ \). We check the satisfiability of \( ρ \) using the available SMT solver.

### 7.3 Constructing a Solution

Algorithm 4 constructs the \( X \) indexed string \( v = R^α (\text{GetS}(θ)) \). More precisely, the algorithm goes through the variables one by one. For each variable \( x ∈ X \) it considers the generic automaton of \( x \) and finds out (i) for each loop \( k: 1 ≤ k ≤ q \), the number \( n_k \) of times the loop is iterated, and (ii) for each state \( i: k(p−p+1) ≤ i ≤ k(p) \) 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 \( i - k \cdot p + p \) if \( a(i) \) occurs a positive number of times. Finally, for a variable \( x \in \mathcal{X} \), we define the string \( v(x) \) by concatenating the strings \( w_k^x \) for all the loops \( k : 1 \leq k \leq q \).

8. Over-Approximation

In this section, we describe the over-approximation module. Fix a finite alphabet \( \Sigma \) and a finite set of variables \( \mathcal{X} \) ranging over \( \Sigma^* \). Suppose that we are given a set \( \psi = \{ \phi_1, \phi_2, \ldots, \phi_k \} \) of constraints together with a set \( \text{Covered} \subseteq \mathbb{N}^2 \) of parameter values that have already been considered by the under-approximation module. In the following, we will construct a set \( \psi' \) of constraints such that \( (\llbracket \psi' \rrbracket - \bigcup_{\alpha \in \text{Covered}} \Gamma^X(\alpha)) \subseteq \llbracket \psi \rrbracket \). To construct \( \psi' \) from \( \psi \), we proceed as follows: (1) we replace any grammar constraint in \( \psi \) by a membership constraint in a regular language that accepts the upward closure of the context-free language \([4, 33] \), (2) we replace any transducer constraint in \( \psi \) by a conjunction of membership constraints in regular languages that capture an over-approximation of the transducer language where each regular language corresponds to the projection of the transducer language on one of its input 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 \( \psi' \) falls in the decidable fragment of the theory of strings with regular membership constraints and length constraints \([2, 3] \). Therefore, we can use a similar technique as the one used in Norn \([2, 3] \) to check the satisfiability of \( \psi' \).

The rest of this section is organised as follows: First, we show how to transform a constraint \( \phi \) appearing in \( \psi \) into a set of constraints \( \text{Over}(\phi) \) in \( \psi' \) using the function \( \text{Over} \). The form of \( \text{Over}(\phi) \) will depend on the type of \( \phi \). Then, we give the precise definition of the set of constraints \( \psi' \) and how to address its satisfiability problem. Finally, we show how to generate new a set of abstraction parameters that will be used by the under-approximation module in case that the set of constraints \( \psi' \) is satisfiable.

**Transforming (dis-)equality constraints.** Let us consider a constraint \( \phi_i \), with \( i : 1 \leq i \leq k \), appearing in \( \psi \). Let us assume that \( \phi_i \) is an (dis-)equality constraint of the form \( x_1x_2 \cdots x_m \sim x_{m+1}x_{m+2} \cdots x_{m+n} \) with \( \sim \in \{ =, \} \). Then \( \text{Over}(\phi_i) \) will only contain the (dis-)equality constraint \( (x_1, i, 1)(x_2, i, 2) \cdots (x_m, i, m) \sim (x_{m+1}, i, m+1)(x_{m+2}, i, m+2) \cdots (x_{m+n}, i, n) \) where we replace any occurrence of a variable \( x \) by a fresh copy of the form \((x, i, j)\).

Let \( \text{Fresh} \) be a function that maps each variable \( x \in \mathcal{X} \) to its set of fresh copies. Formally, the set \( \text{Fresh}(x) \) is the smallest set containing any variable of the form \((x, i, j)\) such that \( \phi_i \) is a (dis-)equality constraint of the form \( x_1x_2 \cdots x_m \sim x_{m+1}x_{m+2} \cdots x_{m+n} \) with \( \sim \in \{ =, \} \), and \( x_j = x \).

**Transforming grammar constraints.** In the following, we will show how to transform a grammar constraint in \( \psi' \) into a set of regular constraints \( \text{Over}(\phi) \) in \( \psi' \) using the function \( \text{Over} \). This transformation is based on replacing the context-free language by a regular language that accepts its upward closure. To do that, we define a function \( \text{Upward} \) that associates for each context-free grammar \( G \) a regular expression \( \mathcal{R} \) such that \( \mathcal{R} \) recognizes the upward closure of the language of the context-free grammar (i.e., \( \mathcal{R} = \{ w' \mid \exists w \in [G] \text{ such that } w \leq w' \} \)). Such a regular expression \( \mathcal{R} \) is effectively constructible from \( G \) \([4, 33] \).

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

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

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

To compute these regular languages, we define a function \( \text{Split} \) that takes as input a transducer \( T \) and outputs a pair of regular expressions \( \mathcal{R}_1 \) and \( \mathcal{R}_2 \) such that \( \{(w_1, w_2) \mid w_2 \in \text{Upward}(w_1)\} \subseteq ([\mathcal{R}_1] \times [\mathcal{R}_2]) \). Let us assume a transducer \( T \) of the form \((Q, \Sigma, \Delta, q_{init}, q_{acc}) \). We can define then two automata \( A_1 = (Q, \Sigma, \Delta_1, q_{init}, q_{acc}) \) and \( A_2 = (Q, \Sigma, \Delta_2, q_{init}, q_{acc}) \) such that \( \Delta_1 \) (resp. \( \Delta_2 \)) is the smallest transition relation containing \((q, a, q') \in \Delta_1 \) (resp. \((q, b, q') \in \Delta_2 \)) if there is a transducer transition of the form \((q, a, b, q') \in \Delta \). Let \( \mathcal{R}_1 \) (resp. \( \mathcal{R}_2 \)) be the regular expression recognizing the same language as \( A_1 \) (resp. \( A_2 \)). We define \( \text{Split}(T) = (\mathcal{R}_1, \mathcal{R}_2) \).

Let us consider a transducer constraint \( \phi_i \) of the form \( y \in \mathcal{G} \). The set \( \text{Over}(\phi_i) \) is defined to be the smallest set containing all the regular constraints of the form \((x, \ell, j) \in \mathcal{R}_1 \) where \((x, \ell, j) \in \text{Fresh}(x) \) and \((y, \ell', j') \in \mathcal{R}_2 \) where \((y, \ell', j') \in \text{Fresh}(y) \).

**Transforming length constraints.** Let us consider a grammar constraint \( \phi_i \) of the form \( \sum_{1 \leq i \leq n} k_i \cdot \text{length}(x_i) = k \), where \( x_i \in \mathcal{X}, k_i \in \mathbb{Z} \) for \( i : 1 \leq i \leq n \), \( k \in \mathbb{Z} \), and \( \sim \in \{ <, =, >, \} \). Then, we define \( \text{Over}(\phi_i) \) to be the smallest set of containing all constraints of the form \( \sum_{1 \leq i \leq n} k_i \cdot \text{length}(x_i) \sim k \) where \((x_i, \ell_i, j_i) \in \text{Fresh}(x_i) \).
Constructing the approximate set of constraints $\psi'$. In order to construct the set of constraints $\psi'$, we need first to construct regular constraints that discard from the set of solutions any string that is accepted by any $\alpha$-flat automaton with $\alpha \in \text{Covered}$. Let $R_{\text{Covered}}$ be the regular expression that accepts the complement of the regular language $\bigcup_{\alpha \in \text{Covered}} F^c_{\alpha}(\alpha)$. We use $\phi_{\text{Covered}}$ to denote the smallest set of constraints of the form $(x, \ell, j) \in R_{\text{Covered}}$ where $(x, \ell, j) \in \text{Fresh}(x)$ for all variable $x \in X$. We define $\psi'$ as $\phi_{\text{Covered}} \cup \overline{\text{Over}}(\phi_1) \cup \overline{\text{Over}}(\phi_2) \cup \overline{\text{Over}}(\phi_3) \cup \cdots \cup \overline{\text{Over}}(\phi_k)$.

Satisfiability problem of the approximate set of constraints $\psi'$. The set of constraints $\psi'$ 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 $\psi'$. Then, let $\overline{\text{Aprx}}(\text{Covered})$ be the algorithm that checks the satisfiability of $\psi'$ and returns a satisfying assignment $v$ for $\psi'$ if $\psi'$ is satisfiable, and 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 $\psi'$. 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 $\text{GenPar}(w)$ to be the set of minimal pairs $(p, q) \in \mathbb{N}^2$ such that there are words $w_1, w_2, \ldots, w_q$ where $\text{length}(w_i) \leq p$ for $i : 1 \leq i \leq q$ and $w \in w_1 \cdot w_2 \cdots \cdot w_q$. Let $\Sigma'$ be the set of variables appearing in $\psi'$. For an $\Sigma'$-indexed string $v$ over $\Sigma$, we define $\text{GenPar}(v)$ to be the maximal pairs in the set $\{ \alpha \mid (x \in \Sigma') \wedge (\alpha \in \text{GenPar}(v(x))) \}$.

9. 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 $\psi$ of string constraints. If the procedure terminates then it either returns an indexed string that satisfies $\psi$, or it concludes that $\psi$ is not satisfiable. The algorithm maintains a set $\text{Covered}$ of parameter values that have already been considered, and a set $\text{Waiting}$ of parameter values to be considered in the coming iterations. The former is initially empty while the latter initially contains the pair $(1, 1)$. The procedure perform alternately a sequence of under- and over-approximation phases. During an under-approximation phase, we check the elements of $\text{Waiting}$ one by one, using the while-loop of line 4. Each time we select and remove a parameter $\alpha$ from $\text{Waiting}$ and move to $\text{Covered}$. We check the under-approximation of $\psi$ wrt. $\alpha$. If the under-approximation produces a satisfying assignment then the procedure terminates. Otherwise, we perform the over-approximation parameterized by the covered set. There are two possible outcomes. If the over-approximation is unsatisfiable then we conclude that $\psi$ is unsatisfiable and we terminate. Otherwise, we get a satisfying assignment $\theta$. In such a case we use $\theta$ to generate a new set of parameters that we put to the waiting set. This will ensure that we at least eliminate $\theta$ in the next iteration, possibly together with an infinite set of other valuations.

10. Experimental Results

We have implemented our framework in our open source solver (called FAT) using Z3 [10] 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 FAT using two separate sets of benchmarks. First, we used the Kaluza benchmarks [27] in order to compare FAT 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 [27] 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 [22] team. The suite consists of approximately 50,000 queries, including length, regular and (dis)-equality constraints.

Figure 4a shows the performance of FAT in comparison with three other state-of-the-art solvers; Z3-str2 [38], CVC4 [21, 22], and S3P [31]. Due to randomness based heuristics, tools may exhibit slightly variable performances. We therefore carry out each experiment three times and only consider the average result. As depicted in Figure 4a, FAT can answer more queries than any of the three other tools. More importantly, it can handle hundreds of queries on which the other solvers reached timeout. These queries were typically

---

Algorithm 5: CEGAR Procedure.

Input: $\psi$: set of word constraints
Output: $\psi$ satisfiable?
1. $\text{Covered} \leftarrow \emptyset$;
2. $\text{Waiting} \leftarrow \{(1,1)\}$;
3. repeat
4. while $\text{Waiting} \neq \emptyset$ do
5. Select and Remove $\alpha$ from $\text{Waiting}$;
6. $\text{Covered} \leftarrow \text{Covered} \cup \{\alpha\}$;
7. $\text{OverAprxResult} \leftarrow \text{OverAprx}(\alpha, \psi)$;
8. if $\text{OverAprxResult} = \text{sat}$ then
9. return $(\psi)$;
10. $\text{AprxResult} \leftarrow \text{Aprx}($Covered$)$;
11. if $\text{AprxResult} = \text{unsat}$ then
12. return (unsat);
13. else if $\text{AprxResult} = \psi$ then
14. $\text{Waiting} \leftarrow \text{GenPar}(\psi)$
15. end if
16. end if
17. end while
18. return $(\text{unsat})$;
19. end if
the largest ones in terms of the number of string variables and of the length of the discovered string solutions. An important hinder for the other solvers on these examples is their use of the arrangement method for solving word equations. The search space explored by the arrangement method is exponential in both the number of variables and in the length of satisfying strings. 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. The running time then indeed grows very quickly when the number of variables on the left and the length of the string on the right increase. Since the running time of our method depends much less 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 [20] 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

```sql
query = "SELECT * FROM records WHERE group=" + groupID + " AND " + userConjunction;
```

can then be 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 \((1 = 1) \lor (1 = 1)\) 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 [30], 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. Note that we supply two columns for FAT: one where we fix an upper bound on the length of the possible solutions, and one were we do not. FAT is the only solver that can handle word equations with length constraints and CFG queries. We compare the performance of FAT to HAMPI which has to bound the length of the possible solutions, and one were we do not. FAT is the only solver that can handle word equations with length constraints and CFG queries. We compare the performance of FAT to HAMPI which has to bound the length of the solutions. Again, observe that FAT is much less affected by the number of variables or by the length of the solutions.

![](image-url)
References


