Efficient Handling of String-Number Conversion

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

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

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

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

Julian Dolby
IBM Research
NY, USA
bui.phi-diep@it.uu.se

Petr Janků
Brno University of Technology
Brno, Czechia
ijanku@fit.vutbr.cz

Hsin-Hung Lin
Academia Sinica
Taipei, Taiwan
hlin@iis.sinica.edu.tw

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

Hsin-Hung Lin
University of Southern California
CA, USA
wwu@isi.edu

Abstract

String-number conversion is an important class of constraints needed for the symbolic execution of string-manipulating programs. In particular solving string constraints with string-number conversion is necessary for the analysis of scripting languages such as JavaScript and Python, where string-number conversion is a part of the definition of the core semantics of these languages. However, the state-of-the-art constraint solvers have limited support to this type of constraint. We propose to overcome these limitations with an approach that can efficiently support both string-number conversion and other common types of string constraints. Experimental results show that it significantly outperforms other state-of-the-art tools on benchmarks that involves string-number conversion.

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

Keywords: String Solver, Formal Verification, Automata

ACM Reference Format:
Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Bui Phi Diep, Julian Dolby, Petr Janků, Hsin-Hung Lin, Lukáš Holík, and Hsin-Hung Lin. 2020. Efficient Handling of String-Number Conversion. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '20), June 15–20, 2020, London, UK*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3385412.3386034

1 Introduction

Symbolic execution is a very popular technique that allows programmers to check the feasibility of a path in a program, i.e., determining the value of the inputs under which the given path can be executed. The path feasibility problem is usually solved by a reduction to the satisfiability of a formula. More precisely, program statements in the path are translated to equivalent constraints in static single assignment (SSA) form and then solved by *Satisfiability Modulo Theory (SMT)* solvers. The types of constraints needed depend on the types of program expressions to be analyzed. Therefore, SMT solvers need to support different combinations of theories so that they can handle a wide range of types.

Among all data types, the *string data type* is omnipresent in modern programming languages. Various security vulnerabilities such as injection and cross-site scripting attack are caused by malicious string values. Therefore, string constraint solving has received considerable attention in the constraint solving community. Operations such as *equality constraints* (e.g. $x.y = y.x$), *regular constraints* (e.g., $x \in (a.b)^*$), and *integer constraints* (e.g., $|x| - |y| > 3$), are widely supported by most state-of-the-art string constraint solvers such as, CVC4 [?], OSTRICH [?], Sloth [?], Trau+ [? ? ?], Z3 [?] and Z3Str3 [?].

An important class of string operations is the string-number conversions. While string length operations are sometimes well supported, converting a string x to an integer n (e.g., using the operation $n = \mathsf{toNum}(x)$) or turning an integer value n into its string form x (e.g., using the operation $x = \mathsf{toStr}(n)$) suffer from extremely limited support by the state-of-the-art string constraint solvers.

In fact, a code that receives string input tends to need to convert at least some of that input into numbers. For example, the program fragment below is a variant of the Luhn test algorithm that is often used in credit card or ID validation.

```
function checkLuhn(value) {
   var sum = 0;
   for (var i = value.length - 1; i >= 0; i-=2) {
       var d = parseInt(value.charAt(i));
       sum += d;
   }
   for (var i = value.length - 2; i >= 0; i-=2) {
       var d = parseInt(value.charAt(i));
       if ((d *= 2) > 9) d -= 9;
       sum += d;
   }
   var last= sum.toString().charAt(sum.length-1);
   return last == '0';
}
```

The input value of the Luhn test algorithm is a sequence of digits. The algorithm processes the digits in the reversed order. The value of every odd digit (e.g., 1st, 3rd, etc.) is added to sum directly. For the value of every even digit, the algorithm (1) doubles its value, (2) subtracts its value by 9 if the doubled-value is larger than 9, and (3) adds the final result to sum. At the end, the input is validated if the last digit of sum is 0 (i.e., `sum mod 10=0`).

To check whether the program path that traverses both loops exactly once and finally passes this test has a valid input, we create the following (string) constraint:

1	$value_0 \in [1,9]^+ \wedge sum_0 = 0 \wedge$
2	$i_0 = \lvert value_0 \rvert - 1 \wedge$
3	$d_0 = \text{toNum}(\text{charAt}(value_0, i_0)) \wedge$
4	$sum_1 = sum_0 + d_0 \wedge$
5	$i_1 = \lvert value_0 \rvert - 2 \wedge$
6	$d_1 = \text{toNum}(\text{charAt}(value_0, i_1)) \wedge$
7	$sum_2 = sum_1 + \text{ite}(d_1 * 2 > 9, d_1 * 2 - 9, d_1 * 2) \wedge$
8	$i_2 = 0$
9	$last_0 = \text{charAt}(\text{toStr}(sum_2), \lvert\text{toStr}(sum_2)\rvert - 1) \wedge$
10	$last_0 = \text{“}0\text{”}$

Here $value_0$ and $last_0$ are string variables and the others are integer variables. The method charAt(x, i) returns the character at index i in the string x while $n = \text{ite}(b, e, e')$ assigns to n the value of the expression e if b is true and the value of the expression e' otherwise. Line 1 describes the initial condition: value should be a sequence of digits and sum is initially zero. Lines 2-4 and lines 5-7 describe one execution of the first and second loop, respectively. Line 8 describes the condition on i_2 before leaving the loop. Finally, Lines 9-10 describe the condition that the last digit of *sum* is zero. Observe that to describe such a program path, we need a solver that supports the following types of constraints:

- *regular* constraints (e.g., $value_0 \in [1,9]^+$, which says $value_0$ is in the regular language $[0,9]^+$),
- *integer* constraints (e.g., $i_0 = \lvert value_0 \rvert - 1$, which says i_0 equals the length of $value_0$ minus one),
- *equality* constraints (often $y = \text{charAt}(x, i)$ is encoded as $x = x_1.x_2.x_3 \wedge \lvert x_1 \rvert = i \wedge \lvert x_2 \rvert = 1 \wedge y = x_2$, which uses equality of string terms x and $x_1.x_2.x_3$), and
- *string-number conversion* (e.g., toStr(sum_2), which is the string value of the number sum_2).

Most of the state-of-the-art string constraint solvers provide only very limited support to the combination of above constraints. Even for simple constraints like the above one, most solvers already fail to provide a correct answer. In Table 3 of our evaluation (Section 9), CVC4 fails to solve constraints corresponding to checkLuhn of more than 6 loop iterations in 2 minutes, Z3 fails to solve the cases corresponding to 4,5,7, and 9 loop iterations, and Z3Str3 fails to solve even the case of 2 loop iterations. Therefore, there is an urgent need for developing efficient techniques for solving such combinations of constraints.

Even, more crucially, in many programming languages, string-number conversion is a part of the definition of their core semantics. JavaScript, which powers most interactive content on the Web and increasingly server-side code with Node.js, is one of such languages. Other scripting languages do too, but we focus on JavaScript due to its prominence. To see how string-integer conversion pervades semantics, consider the following program:

```
for(var i = 0; i < 10; i++) {
    arr[i] = 0;
}
```

A casual glance at the above code reveals no use of strings at all, but the semantics of field access is somewhat unusual in JavaScript: the arrays are indexed by strings, and numeric indices are converted to strings. This conversion is mandated explicitly by the JavaScript semantics: the 2019 edition of ECMAScript [?] requires that *ToPropertyKey* be called on the element expression (§12.3.2.1), and *ToPropertyKey* calls `ToString` on that value in all but special cases (§7.1.14). Therefore, any faithful symbolic execution of JavaScript must handle such conversions for even basic array operations to work correctly. Consider the following code snippet that manipulates an array x, with its value shown on the right:

1	`x = [0, 0, 0, 0, 0]`	[0, 0, 0, 0, 0]
2	`x[3] = 4`	[0, 0, 0, 4, 0]
3	`x[03] = 2`	[0, 0, 0, 2, 0]
4	`x["3"] = 5`	[0, 0, 0, 5, 0]
5	`x["03"] = 7`	[0, 0, 0, 5, 0] and x["03"] = 7
6	`x["03"-1] = 2`	[0, 0, 2, 5, 0] and x["03"] = 7

Here `x[3]` in line 2, `x[03]` in line 3, and `x["3"]` in line 4 all denote the same array element of `x["3"]` (due to the implicit conversion of numeric indices to strings in JavaScript), but `x["03"]` denotes a completely different element (which is stored at the index `"03"` of the array). So naïve modeling of array indices with integers will not work – it cannot distinguish the indices `"3"` and `"03"`.

But if array indices are modeled as strings, we must handle arithmetic somehow. Let us look at the case of line 6, we need to update the value of `x["03"-1]`. The evaluation of the expression `"03"-1` involves an implicit type conversion from the string `"03"` to an integer value 3 due to the `-` (minus) operation. The result of the evaluation of `"03"-1` is the integer 2, which is then converted back to string `"2"` and used as the array index. Hence `x["03"-1]` means the array element of `x["2"]`. Even for a simple example like this, the conversion between string and number is unavoidable. This is a rather basic array operation in JavaScript, and not handling string-number conversion operations will cripple any analysis of non-trivial JavaScript code. Thus, we need stronger solvers that are able to handle string-number conversion operations in order to be able to analyze real code.

Solving string constraint with string-number conversion is a very challenging problem. From the theoretical point of view, this problem is already proven to be undecidable [?]. From practical point of view, our experimental results (in Section 9) show that the current the state-of-the-art string constraint solvers provide only limited support to string-number conversion.

In this paper, we propose a framework that efficiently handles string constraints with string-number conversion. Since the problem is provably unsolvable, our framework combines over and under-approximation techniques. The over-approximation is for proving UNSAT when possible, while the under-approximation is for proving SAT when possible. Both over- and under-approximation fall in a decidable fragment of string constraints that we can efficiently solve.

For ease of presentation, we use the following toy example

$$\Phi = \{xy = yx, n = \mathsf{toNum}(x), n > 3, |y| > |x|, y \in (12)^+\}$$

to explain the main ideas behind our decision procedure. To make our terminology explicit: Φ states that x concatenated with y is the same as y with x, n is the numeric value of the string x, $n > 3$, y is longer than x, and y is a string in the form of a finite repetitions of "12". Notice that Φ is satisfiable. E.g., it has a model x = "12" and y = "1212".

Our decision procedure has two steps: the first step consists in over-approximating the set of input constraints into a set that falls in the chain-free fragment [?], which is decidable. Observe that we could over-approximate the input constraint into any decidable fragment, e.g. the acyclic fragment [?] or the straight-line fragment [?]. Our choice of the chain-free fragment [?] is only motivated by the fact that the chain-free fragment is the *largest* known decidable fragment for that class of string constraints. In our example, we over-approximate the formula Φ by converting $xy = yx$ to two formulae $\{x_1 = xy, x_2 = yx\}$ and replacing the constraint $n = \mathsf{toNum}(x)$ with $n = -1 \vee (n \neq -1 \wedge x \in [0-9]^*)$. Observe that if the over-approximation is UNSAT then our decision

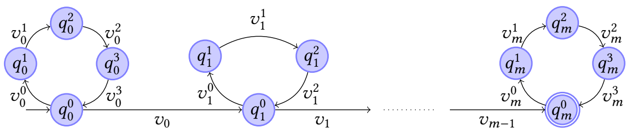


Figure 1. An example of a parametric flat automaton

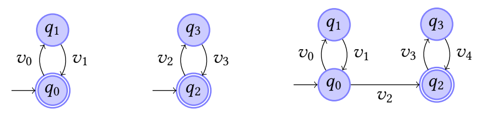
procedure declares that the original formula is also UNSAT and terminates. Surprisingly, despite its simplicity, our over-approximation procedure works very well in practice as shown by our experimental results (in Section 9). Coming back to the formula Φ, the over-approximation module will return SAT in this case.

The second step of our decision procedure is only enabled if the over-approximation step returns SAT. In this case, our decision procedure uses an under-approximation technique (which is our main contribution) to restrict the search domain of each string variable to strings that obey some predefined and parameterized pattern. We propose to use patterns defined by *parametric flat automata* (PFA). A PFA is a *flat* finite state automaton consisting of a predefined sequence of loops, each of fixed length (see Figure 1). The size of the PFA is parameterized by the length of the sequence of loops and the size of each loop. Adjusting these parameters enlarges or prunes the potential solution space. This approach based on PFA is very flexible yet allows very efficient manipulation. In fact, our procedure restricts the search space for each variable to the set of words accepted by the corresponding given PFA.

Then, we show that given such restriction, one can reduce the string constraint solving problem to a linear formula satisfiability problem in polynomial-time. To gain in efficiency, we label each transition of a PFA with a unique *character* variable (whose domain is the set of natural numbers) instead of having a transition between every two states for each symbol in the alphabet. This done by associating to each character in our alphabet a unique natural number. This allows us to avoid the *alphabet explosion problem* from which the approach in [?] suffers and it is the key for handling string-number conversion efficiently.

In the following, we explain the construction of the linear formula using Φ as an example. Assume that we project the domains of x and y to the PFA in Figure 2 (a) and (b), respectively. The variables v_1, v_2, v_3, v_4 in the figure are *character* variables. Thus, v_1, v_2, v_3, v_4 are also integer variables. For example, from $y \in (12)^+$, we may derive $v_2 = 1$ and $v_3 = 2$.

The linear formula produced after the domain restriction will be over variables v_1, v_2, v_3, v_4, as well as the number of occurrences of each character variable $\#v_1$, $\#v_2$, $\#v_3$, $\#v_4$. Each model of the linear formula encodes a model of the string constraint. For example, x = "12" and y = "1212" is encoded by the assignment $(v_1, v_2, v_3, v_4, \#v_1, \#v_2, \#v_3, \#v_4) \rightarrow$



(a) A_x (b) A_y (c) A'_x

Figure 2. Parametric flat automata of x and y

$(1, 2, 1, 2, 1, 1, 2, 2)$.[1] The assignment says, for example, that x is the *parametric word* obtained by traversing the loop of A_x once (because $\#v_1 = \#v_2 = 1$), which is $v_1 v_2$. Under the assignment $v_1 = 1$ and $v_2 = 2$, we obtain $x =$ "12".

If a model of the produced linear formula is found, then the procedure concludes SAT with an assignment to the string variables. If not, our procedure changes the PFAs to a more expressive one (by adding more states and transitions) and repeat the analysis. We report unknown after failing to prove SAT using a certain number of PFAs.

To demonstrate the usefulness of our approach, we have implemented our decision procedure in an open source solver, called Z3-Trau and evaluated it on a large set of benchmarks obtained from literatures and from symbolic execution of real world programs. The experimental results show that Z3-Trau is among the best tools for solving basic string constraints and significantly outperforms all other tools on benchmarks with string-number conversion constraints. In this benchmark, the total amount of tests cannot be solved by Z3-Trau is only a half to the second best tool.

Summary of the Contributions

- An *efficient* procedure for checking satisfiability of string constraints with string-number conversion.
- The class of *parametric flat automata* which is the key for efficient handling of string constraints.
- An algorithm that translates the satisfiability problem of string constraints to the satisfiability problem of a linear formula in polynomial-time when the search space restricted by PFAs.
- An open source tool Z3-Trau with experimental results that demonstrate the efficiency of our approach on both existing and real-life benchmarks

Outline. After recalling the definition in Section 3. Section 4 presents a brief overview of our decision procedure. Section 5 introduces the class of parametric flat automata. Section 6 describes how to use PFA to restrict the searching domain of string variables. Section 7 shows how to construct the linear formula for basic string constraints (i.e., regular, equality, and integer constraints). Section 8 presents the construction of the linear formula for string-number conversion operations. Section 9 presents the details of our implementation and our experimental results. Related works are discussed in Section 10. Finally, Section 11 concludes the paper with a discussion of future works.

[1]In these examples, we use the shorthand $(x_1, \ldots, x_k) \rightarrow (n_1, \ldots, n_k)$ to denote the function $\{x_1 \mapsto n_1, \ldots, x_k \mapsto n_k\}$.

2 Preliminaries

We use $\mathbb{N}$ and $\mathbb{Z}$ to denote the sets of natural numbers and integers. For a set A, we use $|A|$ to denote its size. For $n, m \in \mathbb{N}$, we write $[n, m]$ for the set of natural numbers $\{k \mid n \leq k \leq m\}$. The function f with the domain restricted to a set D is denoted by f_D, and a set of functions F restricted to a set D is $F_D = \{f_D \mid f \in F\}$. An *alphabet* is a finite set Σ of *characters* and a *word* over Σ is a sequence $w = a_1 \ldots a_n$ of characters from Σ, with ϵ denoting the *empty word*. We use $w_1 \cdot w_2$ to denote the *concatenation* of words w_1 and w_2. Σ^* is the set of all words over Σ, $\Sigma^+ = \Sigma^* \setminus \{\epsilon\}$ and $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$. A *language* over Σ is a subset L of Σ^*. We use $|w|$ to denote the length of w and $|w|_a$ to denote the number of occurrences of the character $a \in \Sigma$ in w.

A *finite automaton* (FA) is a tuple (Q, T, Σ, q_i, q_f), where Q is the set of *states*, $T \subseteq Q \times \Sigma \times Q$ is the set of *transitions*, Σ is the alphabet, q_i is the *initial state*, and q_f is the *final state*. A *run* π of A over a word $w = a_1 \cdots a_n$ is a sequence of transitions $(q_0, a_1, q_1), (q_1, a_1, q_2), \ldots, (q_{n-1}, a_n, q_n)$. The run π (resp. the word w) is *accepting* (resp. *accepted*) if $q_0 = q_i$ and $q_n = q_f$. The *language* of A (denoted by $L(A)$) consists of the set of all accepted words.

Through the paper, we will use quantifier-free linear integer arithmetic formulae, and call them *linear formulae* for short. Given a linear formula ϕ over variables V and an *integer interpretation* of V, a function $I : V \longrightarrow \mathbb{Z}$, we denote by $I \models \phi$ that I satisfies ϕ (which is defined in the standard manner), and call I a *model* of ϕ. We use $[\![\phi]\!]$ to denote the set all models of ϕ.

The *Parikh image* of a word $w \in \Sigma^*$ maps each *Parikh variable* $\#a$, with $a \in \Sigma$ is a character, to the number of occurrences of a in w. Formally, given a set S, let $\#S$ denote the set of Parikh variables $\{\#s \mid s \in S\}$. The Parikh image of w is a function $\mathbb{P}(w) : \#\Sigma \longrightarrow \mathbb{N}$ such that $\mathbb{P}(w)(\#a) = |w|_a$. The Parikh image of a language L is defined as follows $\mathbb{P}(L) = \{\mathbb{P}(w) \mid w \in L\}$. It is well known that the Parikh image of a regular language can be characterized by a linear formula:

Lemma 2.1 ([?]). *Let A be a FA over the alphabet Σ. Then, we can compute, in linear time, a linear formula $\Phi_{\mathbb{P}}(A)$, over $\#\Sigma$, such that $[\![\Phi_{\mathbb{P}}(A)]\!]_{\#\Sigma} = \mathbb{P}(L(A))$.*

3 String Constraints

In this section, we formally define string constraints. To begin with, we fix a finite alphabet $\Sigma \subseteq \mathbb{N}$. Note that here we assume that the alphabet is a finite subset of natural numbers. Essentially, we try to capture the numerical encoding of the corresponding symbols in computers (e.g., in ASCII, 'A' is encoded as 65). Hence, we can assume w.l.o.g. that there is a one-to-one mapping between numbers in Σ and the character it encodes. For the simplicity of presentation, we assume that the character '0' is mapped to the number 0, '1' to 1,. . ., and '9' to 9. For other character c, we use $[\![c]\!]$ to denote the number that it maps to. Notice that this approach is general enough to support any finite set of characters.

A minor technical difficulty is that sometimes we may need to treat ϵ as a number. Therefore, we encode ϵ as some fixed number $[\![\epsilon]\!] \in \mathbb{N} \setminus \Sigma$.

Assume that X ia a set of *string variables* ranging over Σ^* and Z a set of *integer variables* ranging over $\mathbb{Z}$. An *interpretation over X and Z* is a mapping $I : X \cup Z \to \Sigma^* \cup \mathbb{Z}$. A *word term* is an element in X^*. We lift the interpretation I to word terms and linear constraints in the standard manner.

We will use four types of *atomic string constraint*:

• An *equality constraint* ϕ_e is of the form $t_1 = t_2$ where t_1, t_2 are word terms. The *model* of ϕ_e is the set of interpretations $[\![\phi_e]\!] = \{I \mid I(t_1) = I(t_2)\}$. A *disequality constraint* ϕ_d is of the form $t_1 \neq t_2$ and is interpreted analogously.

• An *integer constraint* ϕ_i is a linear constraint over the integer variables in Z and values of $|x|$ for all $x \in X$, where $|\cdot| : X \to \mathbb{N}$ is the string length function defined in the standard way. We define $[\![\phi_i]\!] = \{I \mid I(\phi_i) = \text{true}\}$.

• A *regular constraint* ϕ_r is of the form $x \in L(A)$ where x is a string variable and A is a finite automaton. The *model* of ϕ_m is the set of interpretations $[\![\phi_m]\!] = \{I \mid I(x) \in L(A)\}$.

• A *string-number conversion constraint* ϕ_s is of the form $n = \mathsf{toNum}(x)$, where the function $\mathsf{toNum}(x)$ is defined as follows. For $a \in [0, 9]$, we have $\mathsf{toNum}(a) = a$ and for $w \cdot a \in [0, 9]^+$, $\mathsf{toNum}(w \cdot a) = 10 \times \mathsf{toNum}(w) + a$. For $w \notin [0, 9]^+$, $\mathsf{toNum}(w) = -1$. We define $[\![\phi_s]\!] = \{I \mid I(n) = \mathsf{toNum}(I(x))\}$. The *number-string* conversion constraint $x = \mathsf{toStr}(n)$ is treated as a syntactic sugar for $n = \mathsf{toNum}(x)$.

A *string constraint* is then a conjunction of atomic string constraints, with the semantics defined in the standard manner. It is *satisfiable* if there is an interpretation which evaluates the constraint to true. Often we refer to the first three types of atomic string constraints the *basic string constraints*.

Notice that only positive integer is supported in the string-number conversion function. This is the semantics used by most of the SMT solvers, and hence we follow it in this paper. The encoding has a benefit that it can also handle the case where x is "not a number", using the condition $\mathsf{toNum}(x) = -1$. Supporting only positive integer is not a strong restriction, since conversing from negative integer can still be encoded using only the positive version.

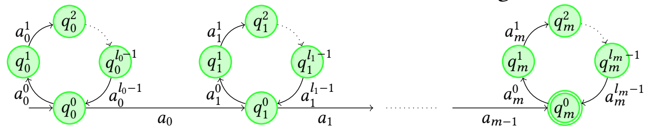
4 Decision Procedure Overview

Our decision procedure has two steps: the first step consists in over-approximating the set of input constraints into a set that falls in the chain-free fragment [?], which is decidable. The over-approximation module proceeds as follows: First, it replaces all string-number conversion constraint $n = \mathsf{toNum}(x)$ by $n = -1 \vee (n \neq -1 \wedge x \in [0-9]^*)$ to obtain an over-approximation Φ consisting of only basic string constraints. Then, it over-approximates Φ to a *chain-free string constraint* [?], which consists of only integer, membership, and *chain-free* equality constraints. Informally, a set of equality constraints has a *chain* if we can find some circular dependency between the string variables in the equality constraints. Our procedure iteratively searches for such dependency chains in the equality constraints. If a chain is found then we replace a variable appearing in that chain by a fresh one. By doing this, we break that chain. We repeat this procedure until there are no more chains. Observe that if the over-approximation is UNSAT then our decision procedure declares that the original formula is also UNSAT.

The second step of our decision procedure is only enabled if the over-approximation step returns SAT. In this case, our decision procedure under-approximates the string constraints by restricting the search domain of each string variable to the language defined by some PFA. This approach based on PFA allows very efficient manipulation. We will show that given such restriction, one can reduce the string constraint solving problem to a linear formula satisfiability problem. The rest of the paper will be mainly dedicated to the explanation of the under-approximation technique (which is our main contribution).

5 Parametric Flat Automata

We introduce *parametric flat automata* that will be used to define patterns used by the under-approximation module to restrict the domain of string variables.



Flat automata. A finite state automaton $A = (Q, T, \Sigma, q_0^0, q_m^0)$ is said to be *flat* if it satisfies the following structural constraints (see also the figure above):

1. The final state q_m^0 is reached from the initial state q_0^0 through a straight path of $m - 1$ transitions $(q_i^0, a_i, q_{i+1}^0) \in T$, $q_i^0 \in Q$ for $i \in [0, m-1]$.
2. Each state q_i^0 is the origin of a unique simple cycle of the length $l_i \in \mathbb{N}$, consisting of states $q_i^{j-1} \in Q$ and transitions $(q_i^{j-1}, a_i^{j-1}, q_i^{j \bmod l_i})$ for $j \in [1, l_i]$. Notice that the case when $l_i = 0$ is also admissible and means that there is no cycle on q_i.
3. Each character in Σ appears on at most one transition.

The crucial feature of flat automata is that their semantics can be faithfully represented by a linear formula and handled efficiently by an SMT solver. Such encoding into linear formula results in efficient algorithms and decision procedures. For instance, we avoid dealing with costly standard automata operations (e.g., checking the non-emptiness of the intersection of several regular languages is known to be PSPACE-complete while it is in NP for the class of flat automata). The encoding is possible due to the flat structure, which has the property that "every word $w \in L(A)$ is uniquely determined by its Parikh image $\mathbb{P}(w)$". More precisely, the Parikh image of a word $w \in L(A)$ can be seen as an encoding of w and can be uniquely decoded:

Lemma 5.1. *For an flat FA A, there is a function $decode_A$ such that for each $w \in L(A)$, $decode_A(\mathbb{P}(w)) = w$.*

Observe that the $\mathbb{P}(w)$ value of any variable appearing within the a cycle of A is equal to the number of repetitions of that cycle in the accepting run. This is an immediate consequence of the fact that every variable appears on at most one transition. Thus, the accepting run on w (and so w itself) can be reconstructed from $\mathbb{P}(w)$.

More concretely, the function $decode_A$ can be implemented as follows. Given $I_\# : \#\Sigma \longrightarrow \mathbb{N}$, and assuming that the lengths of the loops of A are $l_0, \ldots, l_m$, $decode_A(I_\#)$ is constructed as the word $w_0 a_0 w_1 \cdots a_{m-1} w_m$ where for each $i \in [0, m]$, $w_i = (a_i^0 \cdots a_i^{l_i - 1})^{\# a_i^0}$ if $l_i > 0$ and $w_i = \epsilon$ if $l_i = 0$.

For example, in the automaton given at the beginning of this section, from $|x|_{a_0} = |x|_{a_1} = \cdots = |x|_{a_{m-1}} = 1$, $|x|_{a_1^0} = |x|_{a_1^1} = |x|_{a_1^2} = 2$ and $|x|_{a_j^i} = 0$ otherwise, we derive that $x = a_0(a_1^0 a_1^1 a_1^2)^2 a_1 \cdots a_{m-1}$.

Parametric (flat) automata. Next, we define *parametric automaton* (PA) as a pair $P = (A, \psi)$ where A is an automaton operating over an alphabet V of *character variables* and ψ is an *interpretation constraint*, a linear formula over V. *Parametric flat automaton (PFA)* is then a parametric automaton whose automaton is flat. See Figure 1 and 2 for examples of PFAs (without interpretation constraints, i.e., $\psi = \text{true}$.).

Parametric automata accept words over V, called *parametric words*, but we still use them as representations of languages over Σ. Namely, words over V are interpreted as words over Σ under an *interpretation of* V, a mapping $I : V \longrightarrow \Sigma_\epsilon$ (recall that $\Sigma_\epsilon \subseteq \mathbb{N}$). For a parametric word $x = v_1 v_2 \cdots v_k$ over V, its interpretation $I(x)$ is then defined as $I(v_1) \cdot I(v_2) \cdot \ldots \cdot I(v_k)$. We then define the semantics of the PA P as the set of strings $[\![P]\!] = \{I(x) \mid x \in L(A), I \in [\![\psi]\!]\}$ of all interpretations satisfying ψ of all parametric strings in the language of A.

We say that a mapping $I_e : V \cup \#V \longrightarrow \mathbb{N}$ is a *word encoding* of a word w (or a P-encoding of w) if w is an instantiation of some parametric word $x \in L(P)$ whose Parikh image and interpretation of character variables are defined by I_e. Conversely, w is a *P-decoding* of I_e. We use $encode_P(w)$ below to denote all P-encodings of a word w, and $decode_P(I_e)$ to denote all P-decodings of a word encoding I_e. Namely,

$$encode_P(w) ::= \{I_e \mid x \in L(P), I(x) = w, I \in [\![\psi]\!], I_e = I \cup \mathbb{P}(x)\}$$

$$decode_P(I_e) ::= \{w \mid x \in L(P), I(x) = w, I \in [\![\psi]\!], I_e = I \cup \mathbb{P}(x)\}$$

Since a word encoding I_e only records the numbers of occurrences of character variables (Parikh image), the same word encoding may be shared by multiple words, as formalized in the definition of $decode_P(I_e)$.

Example 5.2. Let use consider the PFA $P_x = (A_x, \text{true})$ from Figure 2 (a) and let $Y{=}(v_1, v_2, \#v_1, \#v_2)$. Then we have $encode_{P_x}(\text{"}aaa\text{"}) = \{(Y \rightarrow ([\![a]\!], [\![\epsilon]\!], 3, 3), Y \rightarrow ([\![\epsilon]\!], [\![a]\!], 3, 3)\}$ and $decode_{P_x}((Y \rightarrow ([\![a]\!], [\![\epsilon]\!], 3, 3)) = \{\text{"}aaa\text{"}\}$.

If P is a PFA, then by Lemma 5.1, every word encoding $I_e \in encode_P(w)$ can be decoded uniquely to the word w, i.e.

$$\{w\} = decode_P(encode_P(w))$$

Similarly, as stated by the following corollary of Lemma 5.1, Parikh image of parametric words in $L(A)$ paired with character variable interpretations satisfying ψ encode precisely the words in $[\![P]\!]$.

Corollary 5.3. *For a PFA $P = (A, \psi)$,*
$$[\![P]\!] = decode_P(\{(I \cup I_\#) \mid I_\# \in \mathbb{P}(L(A)), I \in [\![\psi]\!]\}).$$

6 Flat Domain Restriction

In this section, we describe formally how to restrict the domain of string variables to patterns defined by PFA. We start the description of the algorithm that converts a string constraint ϕ_{in} to a linear formula representing the set of solutions under the domain restriction.

The domain restriction is formally defined by restricting the domain of each string variable by a chosen PFA. Namely, assuming that X is the set of string variables of ϕ_{in}, a *flat domain restriction* for ϕ_{in} is a mapping $\mathcal{R}$ that assigns to each variable $x \in X$, a PFA $\mathcal{R}(x)$ over character variables V_x. Let $V_\mathcal{R} = \bigcup_{x \in X} V_x$ be the set of all character variables used in $\mathcal{R}$. We require that these PFA operate over pairwise disjoint sets of character variables, that is if $x \neq y$ then $V_x \cap V_y = \emptyset$. The particular choice of a PFA for each variable depends on the strategy used in the implementation, and will be discussed in Section 9. The *flattening* of the input string constraint ϕ_{in}, denoted $flatten_\mathcal{R}(\phi_{in})$, will be built inductively following the structure of ϕ_{in}. For a conjunction of string constraints, we let $flatten_\mathcal{R}(\phi \wedge \phi') ::= flatten_\mathcal{R}(\phi) \wedge flatten_\mathcal{R}(\phi')$. We do such decomposition until reached atomic string constraints. We show how to build a flattening $flatten_\mathcal{R}(\phi)$ for every atomic string constraint ϕ in the following sections.

The semantics of a string constraint ϕ restricted by $\mathcal{R}$ is then defined as $[\![\phi]\!]^\mathcal{R} = \{I \in [\![\phi]\!] \mid \forall x \in X : I(x) \in [\![\mathcal{R}(x)]\!]\}$.

The correctness of the entire construction of $flatten_\mathcal{R}(\phi_{in})$ is expressed by Theorem 6.2. It uses the decoding function $decode_\mathcal{R}$ parameterized by the domain restriction $\mathcal{R}$. Let Z be the set of integer variables in ϕ_{in}. The function maps an interpretations I_e over $Z \cup V_\mathcal{R} \cup \#V_\mathcal{R}$ to an interpretation over $Z \cup X$, following the domain restriction $\mathcal{R}$. Informally, it "decodes" an interpretation of integer variables $Z \cup V_\mathcal{R} \cup \#V_\mathcal{R}$ to an interpretation of variables in the string constraint ϕ_{in}. Formally, we define $decode_{\mathcal{R}}(I_e) ::= \{I \mid \forall z \in Z : I(z) = I_e(z) \wedge \forall x \in X : \{I(x)\} = decode_{\mathcal{R}(x)}((I_e)_{V_{\mathcal{R}(x)} \cup \# V_{\mathcal{R}(x)}})\}$. The condition says that (1) I and I_e are consistent over variables in Z and (2) $(I_e)_{V_{\mathcal{R}(x)} \cup \# V_{\mathcal{R}(x)}}$ is a word encoding that $\mathcal{R}(x)$-encodes $I(x)$. We also define the $\mathcal{R}$-encoding function as the counterpart of $\mathcal{R}$-decoding, namely, for a interpretation I of the string constraint ϕ_{in}, we let $encode_{\mathcal{R}}(I) = \{I_e \mid decode_{\mathcal{R}}(I_e) = \{I\}\}$. We lift $decode_{\mathcal{R}}$ and $encode_{\mathcal{R}}$ to sets of interpretations in the standard manner.

Example 6.1. We consider the domain restriction $\mathcal{R}$ such that $\mathcal{R}(x) = (A_x, \text{true})$ from Figure 2 (a) and $\mathcal{R}(y) = (A_y, \text{true})$ from Figure 2 (b). Let the set of integer variables be $Z = \{v_z\}$ and let $V_{\mathcal{R}(x)} = \{v_0, v_1, v_2, v_3\}$. For the interpretation $I_e = (v_z, v_0, v_1, v_2, v_3, \#v_0, \#v_1, \#v_2, \#v_3) \rightarrow (3, [\![a]\!], [\![b]\!], [\![c]\!], [\![\epsilon]\!], 3, 3, 2, 2)$, we have $decode_{\mathcal{R}}(I_e) = \{(z, x, y) \rightarrow (3, \text{“}ababab\text{”}, \text{“}cc\text{”})\}$.
Conversely, for $I = (z, x, y) \rightarrow (3, \text{“}ababab\text{”}, \text{“}cc\text{”})$ and $Y = (v_z, v_0, v_1, v_2, v_3, \#v_0, \#v_1, \#v_2, \#v_3)$, we have $encode_{\mathcal{R}}(I) =$
$$\left\{ \begin{array}{l} Y \rightarrow (3, [\![a]\!], [\![b]\!], [\![c]\!], [\![\epsilon]\!], 3, 3, 2, 2), \\ Y \rightarrow (3, [\![a]\!], [\![b]\!], [\![\epsilon]\!], [\![c]\!], 3, 3, 2, 2), \\ Y \rightarrow (3, [\![a]\!], [\![b]\!], [\![c]\!], [\![c]\!], 3, 3, 1, 1) \end{array} \right\}$$

Theorem 6.2. $decode_{\mathcal{R}}([\![flatten_{\mathcal{R}}(\phi_{in})]\!]) = [\![\phi_{in}]\!]^{\mathcal{R}}$

The theorem can be proved by a structural induction over ϕ_{in}. However, for the induction step to go through, we will need to guarantee a stronger correspondence of string constraints ϕ and their flattenings $flatten_{\mathcal{R}}(\phi)$ than just the semantic equality $decode_{\mathcal{R}}([\![flatten_{\mathcal{R}}(\phi)]\!]) = [\![\phi]\!]^{\mathcal{R}}$. Particularly, we will need to ensure that $flatten_{\mathcal{R}}(\phi)$ captures exactly all $\mathcal{R}$-encodings of $[\![\phi]\!]^{\mathcal{R}}$ (indeed, notice that if it would be allowed to capture only some of the encodings, then for instance $flatten_{\mathcal{R}}(\phi) \wedge flatten_{\mathcal{R}}(\phi')$ could only underapproximate $[\![\phi \wedge \phi']\!]^{\mathcal{R}}$). The inductive argument needed in the correctness proof of the under-approximation then reads as

$$[\![flatten_{\mathcal{R}}(\phi)]\!]_{V_{\mathcal{R}} \cup V_{\#\mathcal{R}}} = encode_{\mathcal{R}}[\![\phi]\!]$$

In the next sections, we will formulate the corresponding correctness lemma for flattening constructed from each type of atomic string constraints. We note that the restriction of $[\![flatten_{\mathcal{R}}(\phi)]\!]$ to $V_{\mathcal{R}} \cup V_{\#\mathcal{R}}$ here is needed since $flatten_{\mathcal{R}}(\phi)$ will be constructed with some auxiliary variables.

7 Flattening of Basic String Constraints

We will first discuss flattening of the basic string constraints, that is, regular, equality, and integer constraints. We start by two needed operations over PA, synchronization and concatenation.

Synchronization of PAs. We will now discuss a construction of the *synchronization formula* for two PAs P and P'. It is a linear formula $\Psi_{P \times P'}$ that specifies how each word in the semantic intersection $[\![P]\!] \cap [\![P']\!]$ is encoded by P and by P'. More precisely, the models of $\Psi_{P \times P'}$ represent pairs of word encodings I_e and I'_e such that $I_e \in encode_P(w)$ and $I'_e \in encode_{P'}(w)$ (hence $w \in [\![P]\!] \cap [\![P']\!]$).

Particularly, the synchronization formula is built for two PAs $P = ((Q, T, V, q_i, q_f), \psi)$ and $P' = ((Q', T', V', q'_i, q'_f), \psi')$ such that $V \cap V' = \emptyset$. It is extracted from the *asynchronous product* of P and P'. The asynchronous product is an automaton that uses $Q \times Q'$ as the set of states and $V_\epsilon \times V'_\epsilon$ as the alphabet. Every accepting run of $P \times P'$ corresponds to a pair of accepting runs, a run of P over a parametric word x and a run of P' over a parametric word x'. The word accepted by the run of $P \times P'$ induces constraints on the interpretations of I over V and I' over V' under which the two parametric words have the same interpretation, i.e. $I(x) = I'(x')$.

Intuitively, when the product automaton $P \times P'$ takes a transition $((q_1, q'_1), (v, v'), (q_2, q'_2))$, it means the character variable v and v' should be assigned the same value, P moves under v from state q_1 to state q_2 and P' from q'_1 to q'_2 under v'. When $P \times P'$ takes a transition $((q_1, q'_1), (v, \epsilon), (q_2, q'_1))$, it means that the character variable v should be assigned ϵ, P moves under v to q_1, and P' takes no action, since no action is needed to match P's reading of ϵ (hence consumes no symbol from the input word). Symmetrically, P' might read a variable v assigned ϵ and P may stay.

Formally, the asynchronous product automaton is a tuple $P \times P' = (Q \times Q', T_\times, V_\epsilon \times V'_\epsilon, (q_i, q'_i), (q_f, q'_f))$, where the transition relation $T_\times$ is the minimal set satisfying the following:

- If $(q_1, v, q_2) \in T$ and $(q'_1, v', q'_2) \in T'$, then we have $((q_1, q'_1), (v, v'), (q_2, q'_2)) \in T_\times$.
- If $(q_1, v, q_2) \in T$, then for all states $q' \in Q'$, we have $((q_1, q'), (v, \epsilon), (q_2, q')) \in T_\times$.
- If $(q'_1, v', q'_2) \in T'$, then for all states $q \in Q$, we have $((q, q'_1), (\epsilon, v'), (q, q'_2)) \in T_\times$.

The synchronization formula $\Psi_{P \times P'}$ is extracted from $P \times P'$ as follows. Its first part is the Parikh formula $\Phi_{\mathbb{P}}(P \times P')$ of the product, which encodes all accepting runs of $P \times P'$. The second part is a constraint that extracts from a run of $P \times P'$ the corresponding runs of P and of P':

$$\Psi_\# ::= \left(\bigwedge_{v \in V} \#v = \sum_{x' \in V'_\epsilon} \#(v, x') \right) \wedge \left(\bigwedge_{v' \in V'} \#v' = \sum_{x \in V_\epsilon} \#(x, v') \right)$$

Notice that x, x' are either variables or ϵ. Finally, the third part forces the interpretations of the parametric words accepted by P and P' to be the same:

$$\Psi_= ::= \bigwedge_{x \in V_\epsilon, x' \in V'_\epsilon} \#(x, x') > 0 \rightarrow (x = x')$$

The synchronization formula is then the conjunction

$$\Psi_{P \times P'} ::= \Phi_{\mathbb{P}}(P \times P') \wedge \Psi_\# \wedge \Psi_= \wedge \psi \wedge \psi'$$

The correctness of this construction is stated in Lemma 7.1 below. The correctness of the construction of under-approximations of equality constraints and regular constraints in Section 7.1 and Section 7.2 rely on it.

Lemma 7.1. $[\![\Psi_{P\times P'}]\!]_{V\cup V'\cup\#V\cup\#V'} = \{I_e \cup I'_e \mid I_e \in \mathit{encode}_P(w), I'_e \in \mathit{encode}_{P'}(w), w \in [\![P]\!] \cap [\![P']\!]\}$

Informally, the lemma states that the models of $\Psi_{P\times P'}$ encode precisely the pairs of equivalent encodings of words from $[\![P]\!]$ and $[\![P']\!]$, that constitute the intersection $[\![P]\!] \cap [\![P']\!]$. Since that the models of $\Psi_{P\times P'}$ include also an assignment to the auxiliary variables of $(V_\epsilon \times V'_\epsilon) \cup \#(V_\epsilon \times V'_\epsilon)$, the lemma restricts $[\![\Psi_{P\times P'}]\!]$ to $V \cup V' \cup \#V \cup \#V'$.

Notice that if P is flat (or, symmetrically, if P' is flat), then the semantic intersection $[\![P]\!] \cap [\![P']\!]$ can be still decoded from the synchronization formula $\Psi_{P\times P'}$. Namely, due to Corollary 5.3, we have that if P is a PFA, then

$$\mathit{decode}_P([\![\Psi_{P\times P'}]\!]_{V\cup\#V}) = [\![P]\!] \cap [\![P']\!]$$

Concatenation of PFAs. Concatenation of PFAs will be needed when flattening equality constraints. Its implementation is straightforward, connect the final state of the first PFA with the initial state of the second by an ϵ-transition. Since our automata do not allow transition directly labeled by ϵ, the ϵ-transition is labeled by a fresh variable v_ϵ forced by the constraint $v_\epsilon = \epsilon$ to take the value ϵ.

Formally, given PFA $P = ((Q, T, V, q_i, q_f), \psi)$ and $P' = ((Q', T', V', q'_i, q'_f), \psi')$ with $Q \cap Q' = \emptyset = V \cap V'$, their concatenation is the PFA $P \cdot P' = (Q \cup Q', T \cup T' \cup \{(q_f, v_\epsilon, q'_i)\}, V \cup V' \cup \{v_\epsilon\}, q_i, q'_f, \psi \wedge \psi' \wedge v_\epsilon = \epsilon)$ where v_ϵ is fresh, not from $V \cup V'$.

Lemma 7.2. *$\mathit{encode}_{P.P'}([\![P \cdot P]\!])_{V\cup V'\cup\#V\cup\#V'} = \{I_e \cup I'_e \mid I_e \in \mathit{encode}_P([\![P]\!]) \wedge I'_e \in \mathit{encode}_{P'}([\![P']\!])\}$, for PFAs P and P'.*

With synchronization and concatenation of PA, we are ready to describe flattening of the basic string constraints.

7.1 Flattening of Regular Constraints

Let us first describe the construction of $\mathit{flatten}_{\mathcal{R}}(\phi_r)$ for a regular constraint $\phi_r ::= x \in L(A)$. In order to synchronize the FA A with the PFA $\mathcal{R}(x)$, we represent A by a PA $P' = (A', \Psi_{char})$. The automaton A' of P' operates over fresh character variables $v_a, a \in \Sigma_\epsilon$, and is obtained from A by replacing every occurrence of each character $a \in \Sigma_\epsilon$ on a transition by the variable v_a. The interpretation restriction formula Ψ_{char} of P' then binds the fresh character variables to the character values they represent, namely, $\Psi_{char} = \bigwedge_{a\in\Sigma_\epsilon} v_a = [\![a]\!]$. Obviously, $L(A) = [\![P']\!]$. We then let

$$\mathit{flatten}_{\mathcal{R}}(\phi_r) = \Psi_{\mathcal{R}(x)\times P'}$$

The following lemma states the correctness of this construction. It follows from Corollary 5.3 and Lemma 7.1.

Lemma 7.3. $[\![\mathit{flatten}_{\mathcal{R}}(\phi_r)]\!]_{V_{\mathcal{R}}\cup\#V_{\mathcal{R}}} = \mathit{encode}_{\mathcal{R}}([\![\phi_r]\!])$.

7.2 Flattening of Equality Constraints

We now describe the construction of $\mathit{flatten}_{\mathcal{R}}(\phi_e)$ for an equality constraint $\phi_e ::= x_0 \cdot x_1 \cdots x_n = x_{n+1} \cdot x_{n+2} \cdots x_m$. To simplify the presentation, we assume that the variables are pairwise different, i.e. that $i \neq j \implies x_i \neq x_j$. We may make this assumption without loss of generality, since multiple occurrences of variables in ϕ_e can be eliminated: whenever $x_i = x_j$ for $i \neq j$, we may replace x_j by a fresh variable x'_j and conjoin the modified ϕ_e with a new equality $x_j = x'_j$. We also assume that all disequality constraint $t \neq t'$ are already converted to equivalent equality constraints and integer constraints in the standard way [?].

Having made these assumptions, we may proceed follows. First, we build two PFAs P^{left} and P^{right} that encode the left and the right-hand side word term of the equality constraint ϕ_e, respectively, by concatenating the restrictions of the individual variables. That is

$$P^{left} ::= \mathcal{R}(x_1) \cdot \ldots \cdot \mathcal{R}(x_n)$$
$$P^{right} ::= \mathcal{R}(x_{n+1}) \cdot \ldots \cdot \mathcal{R}(x_m)$$

The under-approximation of ϕ_e is then obtained as their synchronization formula

$$\mathit{flatten}_{\mathcal{R}}(\phi_e) = \Psi_{P^{left}\times P^{right}}$$

The correctness of the construction is stated by the lemma:

Lemma 7.4. $[\![\mathit{flatten}_{\mathcal{R}}(\phi_e)]\!]_{V_{\mathcal{R}}\cup\#V_{\mathcal{R}}} = \mathit{encode}_{\mathcal{R}}([\![\phi_e]\!])$.

7.3 Flattening of Integer Constraints

Given an integer constraint ϕ_l that talks about lengths $|x|$ of string variables $x \in X$. We use a version of ϕ_l where every occurrence of every length function $|x|$ is replaced by an auxiliary length variable l_x and we add a formula to ensure that the value of l_x is equal to that of $|x|$ even when x is encoded using the character variables V_x and Parikh variables $\#V_x$ of $\mathcal{R}(x)$. We will need a set auxiliary variables $\{l_v \mid v \in V_x\}$ to express the length by which the character variable v contributes to the length of an $\mathcal{R}$-encoded string x. That is, l_v will be 0 if v is assigned $[\![\epsilon]\!]$, otherwise it will equal to the number $\#v$ of its occurrences in the word:

$$\Psi_{l_v} ::= (v = [\![\epsilon]\!] \wedge l_v = 0) \vee (v \neq [\![\epsilon]\!] \wedge l_v = \#v)$$

The length of the encoded word x is then the sum of the lengths contributed by the individual character variables in V_x, hence we let

$$\Psi_{l_x} ::= l_x = \sum_{v\in V_x} l_v \wedge \bigwedge_{v\in V_x} \Psi_{l_v} .$$

Finally, the linear formula created for ϕ_l is

$$\mathit{flatten}_{\mathcal{R}}(\phi_l) ::= \phi_l \wedge \bigwedge_{x\in X} \Psi_{l_x}$$

and the following lemma states its correctness.

Lemma 7.5. $[\![\mathit{flatten}_{\mathcal{R}}(\phi_l)]\!]_{V_{\mathcal{R}}\cup\#V_{\mathcal{R}}} = \mathit{encode}_{\mathcal{R}}([\![\phi_l]\!])$

8 Flattening of String-Number Conversion

Last, we present the main contribution of this paper, the construction of a flattening $flatten_{\mathcal{R}}(\phi_s)$ of string-number conversion constraint $\phi_s ::= n = \mathsf{toNum}(x)$.

Let us begin with a simple example. Assume that we use the PFA in Figure 2 (a) to restrict the domain of x. Then we know that when $0 \leq v_0, v_1 \leq 9$, then n is a positive integer value, and otherwise $n = -1$. So we should first add the constraint $((0 \leq v_0 \leq 9) \wedge (0 \leq v_1 \leq 9)) \vee (n = -1 \wedge (v_0 > 9 \vee v_1 > 9))$.

For the case that n is a positive integer, the value of n can be characterized by a constraint (assume character variables are not assigned ϵ) $n = (v_0 \times 10 + v_1) \times (1 + 100 + 100^2 + \ldots 100^{\#v_0 - 1}) = (v_0 \times 10 + v_1) \times \frac{100^{\#v_0} - 1}{100-1}$. The constraint uses $\#v_0$ to capture the total number of loop traversals. Notice that the constraint above contains an exponential component $\frac{100^{\#v_0}}{100-1}$. To solve the satisfiability of this formula, one needs to solve an exponential constraint.

Let us have a look at another example. If we restrict the domain of x to the PFA in Figure 2 (c), for the case when n is positive, we have the relation $n = (v_0 \times 10 + v_1) \times (1 + 100 + 100^2 + \ldots 100^{\#v_0 - 1}) \times 10 \times 100^{\#v_3} + v_2 \times 100^{\#v_3} + (v_3 \times 10 + v_4) \times (1 + 100 + 100^2 + \ldots 100^{\#v_3 - 1}) = (v_0 \times 10 + v_1) \times \frac{100^{\#v_0 - 1}}{100-1} \times 10 \times 100^{\#v_3} + v_2 \times 100^{\#v_3} + (v_3 \times 10 + v_4) \times \frac{100^{\#v_3 - 1}}{100-1}$. Observe that the formula has multiple exponential components, including $\frac{100^{\#v_0} \times 100^{\#v_3}}{100-1}$ and $\frac{100^{\#v_3}}{100-1}$.

It is not difficult to see from the examples above that, if $\mathcal{R}(x)$ is an arbitrary PFA with m loops, the formula that defines the number n contains at least m exponential components, one for each loop. To the best of our knowledge, the satisfiability problem of integer constraints with a mix of polynomials and exponentials is still open. The problem is difficult even for the case that variables are real numbers. For example, the algorithm in [?] involves a quantifier elimination procedure which is double-exponential to the length of the input formula and hence cannot handle large instances. We therefore do not expect that such constraint can be solved efficiently. Instead, we will define a special form of the flat restriction $\mathcal{R}(x)$ of x that leads to an easier linear formula.

PFA that allows efficient string-number conversion: We will now discuss the special form of $\mathcal{R}(x)$, called *numeric PFA*, which we choose for string variables appearing in string-integer constraints and that leads to efficient under-approximation technique. Besides simple induced linear formulae, we still want single $\mathcal{R}(x)$ to cover as many numerals as possible. We want an "easy" completeness property, that is, (1) the space of all numerals can be covered completely by our special numeric PFAs, (2) these numeric PFAs are generated easily, and (3) each of them covers a large and practically significant portion of numerals (so that satisfiable assignments can be often found within the domain restriction of just few of numeric PFAs). Particularly, we will proceed towards a definition of *numeric PFA* (A^m, ψ^m), $m \in \mathbb{N}$ which covers all numerals that encode integers with m digits.

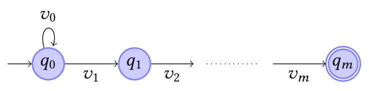
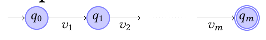
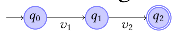


Figure 3. PFA for string-number conversion constraints

Our first attempt is a PFA without loop, i.e., a straight line structure  . The corresponding integer constraint does not have exponential components. However, it does cover all numbers with at most m-digits. Consider the example $\mathsf{toNum}(x) = 10 \wedge |x| = 5$. The number 10 has only 2-digits, at the first glance, a straight-line PFA with two transitions, i.e., the PFA  should be sufficient for the domain restriction of x. If we do so, we will conclude that the formula is unsatisfiable, because the length of x cannot be 5 under this domain restriction.

However, the formula is satisfiable when $x =$ "00010". Observe that toNum("00010") = 10. The key is that even for a bounded integer, the corresponding numeral can be of an unbounded length with arbitrarily many trailing '0's at the front. All numbers with up to m-digits can be however still handled without having to solve exponential constraints. It is enough to equip the initial state of the PFA with a 0-self-loop.

Consequently, the automaton A^m of our number PFA will have the following form illustrated in Figure 3. It has a self-loop on the initial state labeled by the character variable v_0, forced by the constraint

$$\Psi_{v_0} ::= v_0 = 0$$

to hold the value 0. This transition ensures that the under-approximation handles numerals with arbitrary number of trailing zeros. The self-loop is followed by a chain of m transitions (q_{i-1}, v_i, q_i), $1 \leq i \leq m$, leading towards the final state q_m. The chain encodes at most m meaningful digits (only *at most* because the first variables in the sequence can still be assigned zeros and some variables may be assigned ϵ). Hence this PFA covers all numerals that encode numbers with at most m digits. Although it still has a loop, it will not create any exponential component defining the value of n because the loop only represents a sequence of "0" at the front of x. Thus, it will not affect the integer value of $n = \mathsf{toNum}(x)$.

Numeric PFA with these restrictions would satisfy our primary objective, that is, they would induce linear formulae and would "easily" and completely cover all numerals. A last problem still needs to be solved before they can be efficient in practice. Recall that the character variables can be assigned ϵ. Therefore, a single chain of k interesting digits, $k \leq m$, can be by A^m represented in $\binom{k}{m}$ ways, each corresponding to one possible interleaving of k digits with $m-k$ epsilons. This may lead to a formula of exponential size when defining the value of n. In order to eliminate this potential blow-up in the size of the formula, we add to ψ^m an additional constraint that forces all epsilons to be *shifted* behind the least significant digit. This will leave us with only one interleaving. This is the formula

$$\Psi^m_{\mathsf{shift}} ::= \bigwedge_{1 \le i \le m} v_i \ne [\![\epsilon]\!] \implies v_{i-1} \ne [\![\epsilon]\!] \ .$$

Last, since this restriction is meaningful only when the string is indeed a numeral, we also define the constraint representing the strings which are not numerals, the formula

$$\Psi^m_{\mathsf{NaN}} ::= \bigvee_{i \in [1,m]} v_i > 9$$

and define the final form of the interpretation restriction used by A^m as

$$\psi^m ::= \Psi^m_{\mathsf{NaN}} \vee (\Psi_{v_0} \wedge \Psi^m_{\mathsf{shift}}) \ .$$

Consequently, we design our domain restrictions $\mathcal{R}$ so that for string variables x that appear within string-integer constraints, $\mathcal{R}(x)$ is a number PFA (A^m, ψ^m), $m \in \mathbb{N}$.

Assuming that the domain restriction for x is (A^m, ψ^m), the value of the integer n can be extracted from a numeral using the formula

$$\Psi^m_{\mathsf{toInt}} ::= \bigvee_{1 \le k \le m} \Psi^m_{\mathsf{last(k)}} \wedge (n = v_1 * 10^{k-1} + v_2 * 10^{k-2} + \ldots + v_k)$$

where $\Psi^m_{\mathsf{last(k)}}$ says that the last variable of A^m assigned a non-ϵ value is v_k, namely

$$\Psi^m_{\mathsf{last(k)}} ::= (k = m \wedge v_k \ge 0) \vee (v_k \ge 0 \wedge v_{k+1} = -1) \ .$$

Since we also need to distinguish the case when x is not a number, in which case n should equal -1, the formula under-approximating ϕ_s is finally constructed as

$$\mathit{flatten}_{\mathcal{R}}(\phi_s) ::= \Phi_{\mathbb{P}}(A^m) \wedge \\ \big((\Psi^m_{\mathsf{NaN}} \wedge n = -1) \vee (\neg\Psi^m_{\mathsf{NaN}} \wedge \Psi^m_{\mathsf{toInt}})\big)$$

The following lemma states correctness of this construction:

Lemma 8.1. $[\![\mathit{flatten}_{\mathcal{R}}(\psi_s)]\!]_{V_{\mathcal{R}} \cup \# V_{\mathcal{R}}} = \mathit{encode}_{\mathcal{R}}([\![\phi_s]\!])$

9 Implementation and Evaluation

We have implemented our string constraint solving procedure in a tool called Z3-Trau. Z3-Trau is implemented as a theory solver of the SMT solver Z3 [?]. In this way, we can concentrate on solving conjunctive constraints and let Z3 handle the other boolean connectives. Secondly, it makes it possible to solve not only formulae over string constraints but also combinations of string constraints with other theories that Z3 supports. Furthermore, this approach allows us to more effectively handle the arithmetic constraints that are generated by the under-approximation module and eliminate the need to have our own parser for input formulae.

In Z3-Trau, we use the following PFA selection strategy. First, we use *numeric PFAs* for string variables appearing in string-number conversion and *standard PFAs* for others. Then we select size m for numeric PFAs and p loops of size q for standard PFAs. Initially, we set $(m, p, q) = (5, 2, q)$, where q is dynamic and achieved from our internal static analysis. We double m, increase p and q by one if more precise refinement is required. We set an upper bound for each parameter and report UNKNOWN if a solution cannot be found within the bound.

Our over-approximation module uses also some heuristics to derive the constant value of any side of the constraint $n = \mathsf{toNum}(x)$ to refine the over-approximation. For instance, assume in some case, we could derive that the value of $n = 12$ from some integer constraints. Then, we could further derive the value of x is in the regular language (0^*12).

The way our theory solver and Z3 interacts is almost standard. Every time when Z3 asks our theory solver a string constraint satisfiability problem, our solver tries to prove it is SAT or UNSAT using the aforementioned procedure. For under-approximation, every time when a corresponding linear formula is created, we attach the current value of m, p, q with the formula, and then push it to Z3 core. If our theory solver reports UNKNOWN, Z3 remembers it in a global flag `incomplete` and either tries another solution branch, or the same solution branch with different value of m, p, q. If Z3 completes the search of all solution branches, it reports UNSAT if the flag `incomplete` is down, and UNKNOWN otherwise.

We compare Z3-Trau with other state-of-the-art string solvers, namely, CVC4 (45bcf28ab) [?], and Z3 (`d95b549ff`) [?], Z3Str3 (`d95b549ff`) [?]. For these tools, we use the GitHub version stated in the parenthesis because their performance are in general better than the corresponding release version. We do not compare with Sloth [?] since it does not support length constraints, which occurs in most of our benchmarks. We also do not compare with ABC [?] (a model counter for string constraints), Ostrich [?] and Trau+ [?], because they do not support all string functions in our benchmarks, especially string-number conversion.

We perform two sets of experiments. In the first set of experiments, we compare Z3-Trau with other tools on existing benchmarks over basic string constraints. Those benchmarks do not involve string-number conversion function. In the second set of experiments, we compare Z3-Trau with other tools on new suites focusing on string-number conversion. Our goals of experiments are the following:

- Z3-Trau performs as good as, or better than other tools in solving the satisfiability problems of basic string constraints.
- Z3-Trau performs significantly better than other tools in solving the satisfiability problems on string-number conversion benchmark, and this shows the efficiency of PFA in general and *numeric* PFA in particular.

In the first set of experiments, we use the following benchmark examples:

- PyEx [?] is from running the symbolic executor PyEx over some Python packages.
- APLAS [?] includes 600 hand-drafted tests consisting of only equality and integer length constraints.
- LeetCode is from running PyEx over some sample code collected from the LeetCode [?] website, including functions that check if a string is a valid IPv4 or IPv6 address, sum up two binary numbers, check if an input string is an abbreviation of another input string, and decode a sequence of digits to strings according to a given mapping.
- StringFuzz [?] is generated from the fuzz testing tool of the same name.
- cvc4$_{\text{pred}}$ and cvc4$_{\text{term}}$ are obtained from the CVC4 group [?]. These benchmarks contain a small amount of string-number conversion constraints ($< 5\%$).

In the second experiment, we compare with tools supporting string-number conversion on the benchmarks collected from the symbolic executor `Py-Conbyte`[2], which has the supports of string-number conversion. We ran it on several examples collected from the LeetCode platform and from Python core libraries, which involve diverse usages of string-number conversion in Python such as parsing date-time, verifying and restoring IP addresses from strings, etc. We also have examples that encode execution paths of some JavaScript programs (the Luhn algorithm and some array manipulation).

All experiments were executed on a machine with 4-core CPU, 8 GiB RAM. The timeout was set to 10s for each test. We use the results from Z3-Trau, CVC4, and Z3 as the reference answer for the validation of the correctness of the results. Occasionally, two of them report inconsistent (one SAT and one UNSAT) answers to the same test. To decide which solver reports the incorrect answer, we developed a validator, which takes the model I returned from the solver who reported SAT, assigns $I(x)$ to all variables x in the test to obtain a modified test, and re-evaluates the modified test by multiple solvers. If the results from all solvers are consistent, we mark the test SAT or UNSAT according to the results. Otherwise, we manually simplify and inspect the test until we get a conclusive result.

The results of the experiments are summarized in Table **??**, Table 2, and Table 3. Rows with heading `SAT/UNSAT` show numbers of solved formulae. Rows with heading `UNKNOWN`, `TIMEOUT`, or `ERROR` indicate the number of instances the solver fails to return an answer. `INCORRECT` shows the number of cases that a tool gives wrong answer[3].

[2]https://github.com/spencerwuwu/py-conbyte

[3]A correct answer is decided by mainly CVC4, then the consensus of Z3-Trau or Z3.

Table 1. Results of Z3-Trau, CVC4, Z3, and Z3-str3 on String-Number Conversion benchmark.

		Z3-Trau	CVC4	Z3	Z3-str3
Leetcode	SAT	2392	1721	1898	239
	UNSAT	16393	15726	16115	15396
	UNKNOWN	0	0	0	623
	TIMEOUT	142	1480	914	2337
	ERROR	0	0	0	332
	INCORRECT	1	0	0	108
PythonLib	SAT	670	579	914	206
	UNSAT	720	667	724	644
	UNKNOWN	0	0	0	45
	TIMEOUT	1256	1400	1008	1710
	ERROR	0	0	0	41
	INCORRECT	0	0	0	2
JavaScript	SAT	20	6	16	4
	UNSAT	0	0	0	0
	TIMEOUT	0	14	4	16
Total	SAT	3082	2306	2828	449
	UNSAT	17113	16393	16839	16040
	UNKNOWN	0	0	0	668
	TIMEOUT	1398	2894	1926	4063
	ERROR	0	0	0	373
	INCORRECT	1	0	0	110

Table 2. Results of Z3-Trau, CVC4, Z3, and Z3-str3 on String-Number Conversion benchmark.

		Z3-Trau	CVC4	Z3	Z3-str3
Leetcode	SAT	2392	1721	1898	239
	UNSAT	16393	15726	16115	15396
	UNKNOWN	0	0	0	623
	TIMEOUT	142	1480	914	2337
	ERROR	0	0	0	332
	INCORRECT	1	0	0	108
PythonLib	SAT	670	579	914	206
	UNSAT	720	667	724	644
	UNKNOWN	0	0	0	45
	TIMEOUT	1256	1400	1008	1710
	ERROR	0	0	0	41
	INCORRECT	0	0	0	2
JavaScript	SAT	20	3	16	4
	UNSAT	0	0	0	0
	UNKNOWN	0	9	0	0
	TIMEOUT	0	8	4	10
	ERROR	0	0	0	6
	INCORRECT	0	0	0	0
Total	SAT	3082	2306	2828	449
	UNSAT	17113	16393	16839	16040
	UNKNOWN	0	0	0	668
	TIMEOUT	1398	2894	1926	4063
	ERROR	0	0	0	373
	INCORRECT	1	0	0	110

From Table **??**, we can see that the performance of Z3-Trau is as good as the most competitive tools such as CVC4 and Z3 on basic string constraints. In all of the benchmarks, Z3-Trau ranked either the 1st or the 2nd on the number of solved (SAT+UNSAT) cases. If we look into scrutiny, for the APLAS and StringFuzz benchmarks, there is a significant difference between Z3-Trau and the top-ranked tool on the SAT cases. But we believe this is not very important because both these two benchmarks are artificial, either hand-crafted or randomly generated. The more critical ones are those come from program analysis, in which Z3-Trau has comparable performance to the best tool.

From Table 2, we can see that Z3-Trau significantly outperforms all other tools. The total number of failed cases is only a half to Z3, which is ranked the 2nd. In fact, most of the failed tests come from the analysis of one Python core library function that converts an IPv6 address to a number. We generated 2028 tests from this function, and among them around 1000 cases are too difficult for all solvers. If we exclude results on this function, then Z3-Trau has only 218 failed cases in total. This is significantly better than the 2nd place tool Z3, which fails 942 cases in total.

We also ran experiments on the String-Number Conversion benchmark with the timeout set to 30s. Provided more time for solving, Z3-Trau managed to obtain more results and the number of failed cases is reduced to 539, while CVC4 and Z3Str3 got almost the same number of failed cases. Z3 also managed to reduce the number of failed cases to 821.

For further details of the experiment, we have encoded the checkLuhn algorithm introduced in Section 1 for cases of 2 to 12 loops (digits). We ran these tests additionally to the experiments above with the timeout set to 120s. The result is summarized in Table 3. In these tests, Z3-Trau can solve all problems within 1s while CVC4 only returns a model for cases of 2 to 5 loops and Z3Str3 could not solve any of these problems (either timeout or UNKNOWN). However, Z3 can still solve 7 out of the 11 problems and the problems got timeout are cases for 4, 5, 7, and 9 loops. The behavior of Z3 is not unexpected, since all the problems are satisfiable and sometimes the solver are lucky and went into a branch with a correct model very quickly.

Table 3. Comparison of Z3-Trau, CVC4, Z3, and Z3str3 with checkLuhn problems of 2 to 12 loops.

# of Loops	Z3-Trau	CVC4	Z3	Z3Str3
2	**SAT**(0.27s)	**SAT**(0.89s)	**SAT**(0.45s)	**ERROR**
3	**SAT**(0.29s)	**SAT**(1.17s)	**SAT**(0.10s)	**ERROR**
4	**SAT**(0.37s)	**SAT**(4.92s)	**TIMEOUT**	**ERROR**
5	**SAT**(0.39s)	**SAT**(11.27s)	**TIMEOUT**	**ERROR**
6	**SAT**(0.41s)	**TIMEOUT**	**SAT**(0.13s)	**UNKNOWN**
7	**SAT**(0.51s)	**TIMEOUT**	**TIMEOUT**	**ERROR**
8	**SAT**(0.53s)	**TIMEOUT**	**SAT**(0.29s)	**ERROR**
9	**SAT**(0.63s)	**TIMEOUT**	**TIMEOUT**	**ERROR**
10	**SAT**(0.69s)	**TIMEOUT**	**SAT**(0.48s)	**TIMEOUT**
11	**SAT**(0.71s)	**TIMEOUT**	**SAT**(0.36s)	**ERROR**
12	**SAT**(0.74s)	**TIMEOUT**	**SAT**(0.38s)	**TIMEOUT**

10 Related works

To the best of our knowledge, the study of solving string constraint traces back to 1946, when Quine [?] showed that the first-order theory of string equality constraints (a.k.a. word equations) is undecidable. Makanin achieves a milestone [?] by showing that the class of quantifier-free string equality constraints is decidable. Since then, several works, e.g., [? ? ? ? ? ? ?], consider the decidability and complexity of different subclasses of string equality constraints.

Satisfiability of string constraints is a challenging problem. The satisfiability of equality constraints combined with length constraints of the form $|x| = |y|$ is already opened for more than 20 years [?]. Numerous decidable fragments were proposed [? ? ? ? ? ?]. Among them, the chain-free fragment [?] used by our over-approximation module is the largest known decidable fragment, which allows us to produce more precise over-approximation and hence solve many UNSAT instances efficiently.

The strong practical motivation led to the rise of several string constraints solvers that concentrate on solving practical problem instances. Several tools handle string constraints assuming a fixed upper bound on the length of strings and translate them to boolean satisfiability problems [? ? ?]. Our method, on the other hand, allows analyzing constraints without a length limit and still with some completeness guarantees, i.e., within the language defined by PFAs.

More recently, *DPLL(T)-based* string solvers [? ? ? ? ? ? ? ? ?] lift the restriction of strings of bounded length. They usually support a variety of string constraints, including all basic string constraints, and sometimes also regular/rational relations. The typical procedure they used for solving equality constraints is to split them into simpler sub-cases, in combination with powerful techniques for Boolean reasoning to curb the resulting exponential search space. In contrast, our approach uses a completely different search strategy. We restrict the solution space to some predefined pattern and step-wisely enrich the pattern in use.

The most relevant work to ours is [?] who proposed to project the solution space of variables to a generalization of flat automata. The main difference is that our approach works fully symbolically, which is enabled by to the ideas of using character variables and hence PFAs. The use of character variables allows our approach to handle string-number constraints efficiently – the values and number of occurrences of those variables can be directly convert to a number in a linear formula. Moreover, the approach of [?] is exponential to the alphabet size (we call this the alphabet-explosion problem in the introduction), and hence in the implementation, usually requires heuristics to prune out unnecessary symbols.

A further direction is *automata-based* solvers for analyzing string-manipulated programs. ABC [?] and Stranger [?] soundly over-approximates string constraints using multi-tape automata [?], and outperforms DPLL(T)-based solvers when checking single execution traces, according to some evaluations [?]. People also studied the combination of automata-based algorithms with 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 and word equations, cannot be entirely handled by automata-based solvers.

11 Conclusion and Future Works

In this paper, we report a novel approach for solving string constraints with string-number conversion and implemented it as an open-source tool Z3-Trau. For now, it support basic string constraints, string-number conversion, and also operations that can be encoded to them (e.g., CONTAINS, PREFIXOF). Since Z3-Trau is built inside the SMT solver Z3, we also get the power of processing formulae in the combination of different theories (e.g., array). Hence our tool can support the encoding of a wide range of program expressions. There are several avenues for future works. First, we are planning to integrate it with the JavaScript symbolic executor COSETTE [**?**]. We believe such integration is feasible. We are also planning to merge Z3-Trau with the main branch of the Z3 solver. For technical development, we think it would be interesting to consider the (symbolic) flattening of an even larger set of string operations, such as the one containing REPLACEALL and SPLIT.