Cooking String-Integer Conversions with Noodles

Vojtěch Havlena
Brno University of Technology, Czech Republic

Lukáš Holík
Brno University of Technology, Czech Republic

Ondřej Lengál
Brno University of Technology, Czech Republic

Juraj Síč
Brno University of Technology, Czech Republic

Abstract

We propose a method for efficient handling string constraints with string-integer conversions. It extends the recently introduced stabilization-based procedure for solving string (dis)equations with regular and length constraints. Our approach is to translate the conversions into a linear integer arithmetic formula, together with regular constraints and word equations. We have integrated it into the string solver Z3-Noodler, and our experiments show that it is competitive and on some established benchmarks even several orders of magnitude faster than the state of the art.

2012 ACM Subject Classification Theory of computation → Logic and verification

Keywords and phrases string solving, string conversions, SMT solving

Digital Object Identifier 10.4230/LIPIcs.SAT.2024.14

Supplementary Material *Software (Source Code)*: `https://github.com/VeriFIT/z3-noodler` [34] archived at `swh:1:dir:a3be55dc798afb9fa29b90ddb236653e5c13e275`

Funding This work was supported by the Czech Ministry of Education, Youth and Sports ERC.CZ project LL1908, the Czech Science Foundation project 23-07565S, and the FIT BUT internal project FIT-S-23-8151.

1 Introduction

String constraint solving (string solving for short), has garnered significant attention in recent years, particularly due to its important role in verifying web programs. These programs are often written in string-intensive programming languages, such as PHP or JavaScript. Careless handling of strings can inadvertently expose systems to severe security vulnerabilities, such as SQL injection or cross-site scripting (XSS), both of which remain prevalent security risks [29, 28]. Since string constraints establish a general formal framework for working with strings, new applications of string solving still emerge. Notable examples include analyzing user policies within Amazon Web Services [25] and analyzing smart contracts [7].

String solvers are typically integrated into general SMT solvers as theory solvers, enabling the combination of string constraints with other SMT theories. The most mature SMT solvers supporting string constraints are cvc4/5 [8, 31] and Z3 [22]. Furthermore, the original string theory solver can be replaced, leading to string solvers such as Z3str3RE [13], Z3str4 [27], Z3-Trau [3, 2], OSTRICH [16] (based on the Princess SMT solver [32]), and, most recently, Z3-Noodler [18, 17]. Except these general SMT solvers, there are some string-only solvers, such as Norn [5], Kepler$_{22}$ [24], Woorpje [21], or Retro [19, 20].

In order to handle constraints occurring in real-world applications, string solvers need to support not only *basic* constraints, such as string equations, regular membership queries, and length constraints, but also *extended* constraints such as various kinds of string functions and

predicates (e.g., `replace`, `indexof`, `substr`, or `prefix`) or string-integer conversions allowing to identify strings with their numeric values. These string manipulating functions are widely used in particular in the context of verification of web programs, as they correspond to the string operations in programming languages such as PHP or JavaScript.

Complex combinations of various types of basic and extended constraints, which is how they typically appear, is challenging for today's solvers. In this paper, we combine the recently introduced stabilization-based procedure for solving basic string constrains [14, 18] with a novel technique for handling string-integer conversions. The fundamental principle of the stabilization-based procedure is an iterative refinement of regular constraints of string variables, which represent all solutions of the string constraint, until a stable form is reached. In order to handle the string-integer conversion, namely `to_int`, `from_int`, `to_code`, and `from_code` predicates, our procedure performs reasoning over the stabilized regular constraints of string variables used in the conversions. The proposed procedure generates a *linear integer arithmetic* (LIA) formula concisely encoding possible numerical values from strings of a stabilized regular constraint and relate them with their particular lengths. If this LIA formula is satisfiable, so is the original constraint with conversions. To avoid exponential blow-up during the formula generation, we express languages as finite sets of intervals encoding all valid numerical values. Since different conversions may be applied on the same variable/sequence of variables, the generated formula must also relate the values of these conversions. In order to be precise, our procedure requires finiteness of stabilized languages of variables occurring in conversions. For infinite languages, we propose an underapproximation restricting the languages to strings of a particular length.

We implemented the proposed technique into Z3-Noodler [18, 17, 34], a string solver based on Z3 implementing the stabilization-based procedure, and compared our technique with other string solvers on all benchmarks from SMT-LIB containing string-integer conversions. Our experimental evaluation shows that our proposed technique is competitive and on many instances even several orders of magnitude faster than the state of the art.

2 Related work

The decidability of various fragments of string constraints have been studied for a long time. If a string constraint contains only string equations, the satisfiability problem is decidable [26]. Moreover, it is decidable even if we add regular constraints [33]. A question whether the combination of string equations and length constraints is decidable is still open. Adding string-integer conversions to string equations with regular and length constraints leads to undecidability [23]. The undecidability still holds even if we keep only the concatenation instead of full string equations [12].

The tools supporting string-integer conversions therefore implement incomplete procedures. The string solver cvc4/5 implements a derivation-based decision procedure for solving code-point conversions (conversions between characters and their numeric values), which are then used to define a universally quantified formula for handling string-integer conversions [30]. Z3-Trau implements string-integer conversions as a part of the flattening-based procedure [1]. The main differences between the approach to string-integer conversions in Z3-Trau and our proposed technique are twofold: (i) The approach of Z3-Trau is integrated to a string procedure underapproximating the variable languages by flat languages. We combine our approach with the fundamentally different stabilization-based procedure working with arbitrary precise regular languages. (ii) Z3-Trau underapproximates string-integer conversions restricting arguments having at most m digits (m is a parameter). To the contrary, our

conversion approach works with precise regular languages describing all solutions and is complete for finite languages. We only underapproximate if the languages for conversion are infinite. Moreover, we underapproximate the precise languages and not the already underapproximated flat languages as is the case of Z3-Trau. More tools providing a (limited) support for string-integer conversions are Z3 [22], Z3str4 [11], and OSTRICH [16].

3 Preliminaries

Functions, strings and languages. We use $\mathbb{Z}$ to denote the set of integers and $[k_1, k_2]$, $k_1, k_2 \in \mathbb{Z}$, for the set of all integers between k_1 and k_2, including k_1 and k_2. For a set X we use id_X to denote the identity function over X. We fix a finite alphabet Σ (we denote symbols as $a, b, c, \dots$) and for the rest of the paper, we assume that it contains symbols `'0'`, `'1'`, ..., `'9'`. A *string* over Σ is a finite sequence $u = a_1 \dots a_n$ of symbols from Σ. We use ϵ to denote the empty string. To avoid confusion with integers, we sometimes use quotes to denote strings with digits, e.g., `'42'` (we skip it if it is clear from the context). We further use $|u|$ to denote the *length* of u with $|\epsilon| = 0$. The set of all strings is denoted by Σ^*. The *concatenation* of strings u and v is denoted $u \cdot v$, or uv for short (ϵ is the neutral element). A *language* is a subset of Σ^*. The *concatenation* of two languages L_1 and L_2 is defined as $L_1 \cdot L_2 = \{u \cdot v \mid u \in L_1 \wedge v \in L_2\}$. *Bounded iteration* of a string/language x for $i \geq 0$ is defined recursively as (i) $x^0 = \{\epsilon\}$ for the case of languages and $x^0 = \epsilon$ for the case of strings, and (ii) $x^{i+1} = x^i \cdot x$. *Iteration* is then defined as $x^* = \bigcup_{i \geq 0} x^i$ and *positive iteration* is defined as $x^+ = \bigcup_{i \geq 1} x^i$. We denote regular languages using regular expressions RE with the following standard notation:

$$\mathsf{RE} ::= \emptyset \mid a \mid \epsilon \mid (\mathsf{RE}) \mid \mathsf{RE}^* \mid \mathsf{RE}\,\mathsf{RE} \mid \mathsf{RE} + \mathsf{RE} \mid \mathsf{RE}^n$$

where $a \in \Sigma$ and $n \in \mathbb{N}$. We further use RE^+ as a syntactic sugar for $\mathsf{RE}\,\mathsf{RE}^*$ and S to denote the regex $v_1 + \dots + v_n$, where $S = \{v_1, \dots, v_n\}$ is a finite subset of Σ^*.

String constraints. In this paper, we consider *string constraints* over alphabet Σ, string variables $\mathbb{X}$, and integer variables $\mathbb{I}$. The string variables range over Σ^* and integer variables over $\mathbb{Z}$. The syntax of a string contraint φ is given by the following grammar:

$$\begin{aligned}
\varphi &::= t_i \leq t_i \mid t_s = t_s \mid t_s \in \mathsf{RE} \mid \varphi \wedge \varphi \mid \neg\varphi \\
t_s &::= v \in \mathbb{X} \mid t_s \cdot t_s \mid \texttt{from_code}(t_i) \mid \texttt{from_int}(t_i) \\
t_i &::= v \in \mathbb{I} \mid k \in \mathbb{Z} \mid t_i + t_i \mid |t_s| \mid \texttt{to_code}(t_s) \mid \texttt{to_int}(t_s)
\end{aligned}$$

where t_s is a string term consisting of a concatenation of string variables[1] and string-integer conversions, and t_i is a *linear integer arithmetic* (LIA) term containing among usual arithmetic terms also a string-length term and integer conversions of string terms.

Semantic of the conversion functions is then given as follows[2]. The conversions $\texttt{from\_int}(k)$ and $\texttt{to\_int}(s)$ convert between strings and integers: if $k \geq 0$, $\texttt{from\_int}(k)$ is the string representation (without leading zeroes) of the number k and ϵ otherwise, while $\texttt{to\_int}(s)$ returns the *non-negative* number represented by the string s, or -1 if s does not represent a non-negative integer (with leading zeros allowed). For example, $\texttt{from\_int}(5) = \texttt{'5'}$, $\texttt{from\_int}(-6) = \epsilon$, and $\texttt{to\_int}(\texttt{'a'}) = -1$. On the other hand,

[1] Note that an explicit string $u \in \Sigma^*$ can be encoded by a fresh variable x and a regular constraint $x \in u$.
[2] following the definition of the string theory of the SMT-LIB standard [10]

$\texttt{from\_code}(k)$ and $\texttt{to\_code}(s)$ convert between a Unicode symbol and its *code point*. More specifically, if s is a single symbol, then $\texttt{to\_code}(s)$ returns its code point, otherwise it returns -1; $\texttt{from\_code}(k)$ returns the symbol s whose code point is k (and ϵ if k is out of the range of Unicode symbols). For example, $\texttt{to\_code}(\texttt{'0'}) = 48$ and $\texttt{from\_code}(48) = \texttt{'0'}$.

An *assignment* ν is a mapping $\nu\colon (\mathbb{X} \to \Sigma^*) \cup (\mathbb{I} \to \mathbb{Z})$. The value of a term in ν is defined as usual. An assignment ν is a *model* of an atomic constraint $t = s$ iff $\nu(t) = \nu(s)$, of $t \in \mathsf{RE}$ iff $\nu(t) \in \mathsf{RE}$, and of $t \leq s$ iff $\nu(t) \leq \nu(s)$. The definition of model is extended to Boolean combinations of constraints as usual.

We will often work with a pair (Lang, σ), where $\mathsf{Lang}\colon \mathbb{X} \to 2^{\Sigma^*}$ is a *language assignment* assigning a language to each string variable and $\sigma\colon \mathbb{X} \to \mathbb{X}^+$ is a *substitution*. We use $\sigma(\Phi)$ to denote the string constraint where every occurrence of every $x \in \mathbb{X}$ is replaced by $\sigma(x)$. We define the composition of two substitutions σ_1 and σ_2 as $\sigma_1 \circ \sigma_2 = \{x \mapsto \sigma_1(\sigma_2(x))\}$. We say that a substitution σ is *flat* if for each string variable x appearing in its image, $\sigma(x) = x$ (hence $\sigma \circ \sigma = \sigma$).

Normalization of string constraints. We assume that our decision procedure is used within a DPLL(T)-based SMT solver, which has the property that theory solvers are always given conjunctions of (possibly negated) atomic constraints (in the given theory). In the rest of the paper, we therefore assume that string constraints have this normal form, more specifically, that they are of the form $\mathcal{E} \wedge \mathcal{R} \wedge \mathcal{L} \wedge \mathcal{C}$ where

- $\mathcal{E}$ is a conjunction of (dis)equations of concatenated *string variables*, i.e., they do not contain conversions $\texttt{from\_int}(t_i)$ and $\texttt{from\_code}(t_i)$. We can remove each such conversion by replacing it with a fresh string variable x and adding a new equation $x = \texttt{from\_int}(t_i)$ (or $x = \texttt{from\_code}(t_i)$) to $\mathcal{C}$.
- $\mathcal{R}$ is a conjunction of regular constraints of the form $x \in \mathsf{RE}$ for $x \in \mathbb{X}$. We can convert any predicate $t_s \in \mathsf{RE}$ into this form by replacing it in $\mathcal{R}$ with $x \in \mathsf{RE}$ and adding $x = t_s$ to $\mathcal{E}$, where x is a fresh variable. Negated regular constraints $x \notin \mathsf{RE}$ can be rewritten into the positive form by complementing the corresponding regular language.
- $\mathcal{L}$ is a LIA constraint without conversions (length constraints are allowed). Again, conversions `to_int` and `to_code` are replaced with a fresh integer variable and a corresponding equation in $\mathcal{C}$.
- $\mathcal{C}$ is a conjunction of equations of the form $x = \texttt{from\_code}(y)$, $x = \texttt{from\_int}(y)$, $y = \texttt{to\_code}(x)$, and $y = \texttt{from\_code}(x)$ for $x \in \mathbb{X}$ and $y \in \mathbb{I}$ (if needed, the generic terms in arguments of conversions are replaced with a fresh variable and a corresponding equation).

Note that in the rest of the paper, we sometimes treat a conjunction of literals (i.e., atomic constraints or their negations) as a set of these literals.

4 Stabilization-based Procedure for String Constraints

We briefly describe the stabilization-based procedure from [14, 18] for solving a *conversions-free* string constraint $\mathcal{E} \wedge \mathcal{R} \wedge \mathcal{L}$ where $\mathcal{E}$ contains only *equations* (no disequations). From a high-level point of view, the stabilization-based procedure iteratively splits equations (where the equation splits are represented by substitution maps) and refines the language assignments until a stable solution is found, which is then used to generate a LIA formula describing lengths of all solutions that this language assignment represents.

Stable solution. We say that (Lang, σ), a pair of a language assignment and a substitution, is a *stable solution* of $\mathcal{E} \wedge \mathcal{R}$ if each language in Lang is nonempty, σ is flat, and every assignment $\nu\colon \mathbb{X} \to \Sigma^*$, such that $\nu(x) \in \mathsf{Lang}(x)$ and $\nu(x) = \nu(\sigma(x))$ holds for all $x \in \mathbb{X}$,

is a model of $\mathcal{E} \wedge \mathcal{R}$. Loosely speaking, the stability means that we can choose arbitrary assignments of the substituted variables from Lang and, using σ, construct a model of the original constraint.

▶ **Example 1.** Consider a string constraint $xy = z \wedge w = xx \wedge w \in a^+$ over the alphabet $\{a, b\}$. Let v_1, v_2 be fresh variables, Lang a language assignment where $\mathsf{Lang}(v_1) = \mathsf{Lang}(w) = a^+$ and all other variables are mapped to $(a + b)^*$, and let $\sigma = \{x \mapsto v_1, w \mapsto v_1v_1, z \mapsto v_1v_2, y \mapsto v_2, v_1 \mapsto v_1, v_2 \mapsto v_2\}$ a substitution. Note that σ is flat and by selecting words for v_1 and v_2, we can get words for the other variables using σ. Furthermore, for an arbitrary selection of words from the languages of v_1 and v_2, the corresponding string assignment is a model of the string constraint. For instance, $\nu = \{x \mapsto a, w \mapsto aa, y \mapsto b, z \mapsto ab, v_1 \mapsto a, v_2 \mapsto b\}$ is a model for a selection of words $v_1 \mapsto a \in \mathsf{Lang}(v_1)$ and $v_2 \mapsto b \in \mathsf{Lang}(v_2)$. Therefore, (Lang, σ) is a stable solution. ⌟

Noodlification. A key notion of the stabilization-based procedure introduced in [14] and later extended to string constraints with lengths [18] is *noodlification*. Noodlification allows to steer the generation of possible equation splits (or alignments) only to feasible ones. An *equation split* is a splitting of variables to a concatenation of new fresh variables so that boundaries between variables on both sides of an equation match. A variable split is performed together with splitting the variable's language in Lang. A feasible split has a property that the fresh variables have nonempty languages. For instance, for the equation $xyu = wz$ with $\mathsf{Lang} = \{x \mapsto a^*, p \mapsto (a + b)^* \mid p \in \{y, u, w, z\}\}$, one possible feasible split is $\{x = v_1v_2, y = v_3, u = v_4, w = v_1, z = v_2v_3v_4\}$ together with the language assignment $\mathsf{Lang}' = \mathsf{Lang} \cup \{v_1 \mapsto a^*, v_2 \mapsto a^*, v_3 \mapsto (a + b)^*, v_4 \mapsto (a + b)^*\}$.

The length-aware noodlification, denoted as $\mathsf{noodlify}(t = s, \mathsf{Lang})$, generates a set of all feasible splits of the equation $t = s$. A split is a pair (E, Lang') where E is a set of new equations and Lang' is a language assignment of fresh variables. An efficient implementation of noodlification uses nondeterministic finite automata (NFAs) to represent the language assignment[3]. The noodlification then constructs ϵ-preserving products $\mathcal{P}$ of ϵ-concatenated automata of t and s. Each side has a different ϵ-symbol serving as a delimiter between variables of each equation side. The resulting feasible splits are then generated by inspecting parts of the product automaton $\mathcal{P}$.

Simplified stabilization-based procedure. In this paragraph we describe a simplified length-aware stabilization-based procedure. We briefly discuss particular optimizations of the procedure introduced in [14, 18] at the end of the paragraph. The stabilization-based procedure starts from the tuple $(\mathcal{E}, \mathsf{Lang}_\mathcal{R}, \mathsf{id}_\mathbb{X})$ where $\mathcal{E}$ are string equations and $\mathsf{Lang}_\mathcal{R}$ is the language assignment obtained from the initial regular constraint $\mathcal{R}$. It then proceeds by applying inference rules given below, trying to reach a tuple $(\emptyset, \mathsf{Lang}, \sigma)$ such that (Lang, σ) is a stable solution (which denotes satisfiability of the original constraint) or a tuple where $\mathsf{Lang}(x) = \emptyset$ for some $x \in \mathbb{X}$ (which denotes unsatisfiability).

The first rule is Align&Split, which performs noodlification on an equation $s = t$ and adds equations representing generated feasible alignments, removing $s = t$ from the set of equations (the formula on the right-hand side of a rule denotes a condition when the rule can be applied):

[3] We start with NFAs obtained from initial regular constraints and in each step perform only regularity-preserving operations.

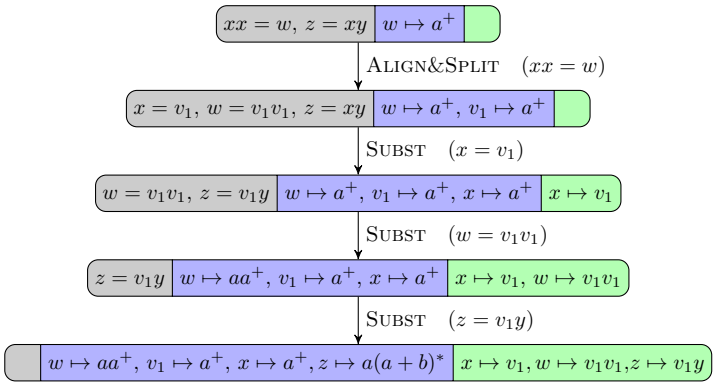


Figure 1 Part of a proof graph generated by rules of simplified stabilization-based procedure for the string constraint $xy = z \wedge w = xx \wedge w \in a^+$ over the alphabet $\{a, b\}$. Tuples $(\mathcal{E}, \mathsf{Lang}, \sigma)$ are visualized by nodes $\boxed{\mathcal{E}}\boxed{\mathsf{Lang}}\boxed{\sigma}$ (we omit there implicit language assignments of variables to $(a+b)^*$ as well as identity substitutions $x \mapsto x$). Edges are labelled by the used rule together with the selected equation. The language assignment and substitution in the bottom node form a stable solution.

$$\textsc{Align\&Split}\colon \frac{(\ \mathcal{E} \uplus \{s = t\},\quad \mathsf{Lang},\quad \sigma\ )}{\big\{(\ \mathcal{E} \cup \mathit{Align}_i,\quad \mathsf{Lang}_i,\quad \sigma\ )\big\}_{i=1}^{n}}\ \mathsf{noodlify}(s = t, \mathsf{Lang}) = \{(\mathit{Align}_i, \mathsf{Lang}_i)\}_{i=1}^{n}$$

The second rule, SUBST, removes a simple equation $x = t$ where $x \in \mathbb{X}$ from the set of equations and transforms it into a substitution, which is then applied to the set of equations.

$$\textsc{Subst}\colon \frac{(\quad \mathcal{E} \uplus \{x = t\},\quad \mathsf{Lang},\quad \sigma,\quad)}{\big\{(\ \sigma_i'(\mathcal{E} \cup \mathit{Align}),\quad \mathsf{Lang}_i,\quad \sigma_i' \circ \sigma\ )\big\}_{i=1}^{n}}\ \varphi_{\textsc{Subst}}$$

where $\varphi_{\textsc{Subst}} \overset{\text{def}}{\Leftrightarrow} \mathsf{noodlify}(x = t, \mathsf{Lang}) = \{(\mathit{Align} \uplus \{x = t_i'\}, \mathsf{Lang}_i)\}_{i=1}^{n} \wedge \bigwedge_{i=1}^{n} \sigma_i' = \{x \mapsto t_i'\}$.

In [14, 18] these basic rules were extended by organizing equations in inclusion graphs, which then determine the order of selecting equations in ALIGN&SPLIT. Moreover, the notion of equations is replaced by inclusions allowing to distinguish sides of equations, which is important for the completeness on the *chain-free fragment* of string constraints [18, 6]. The extended rules also take into account information about length variables, which are propagated through alignments. During the concatenation of automata corresponding to non-length variables, ϵ symbols may be removed, leading to a significant reduction of the number of splits. Additional reduction of the number of generated splits during noodlification is achieved by eager simulation-based reduction applied on NFAs in the language assignment.

Generating length formulae. In order to check that a stable solution (Lang, σ) is satisfiable w.r.t. the initial length constraint $\mathcal{L}$, we first generate for each $x \in \mathbb{X}$ a LIA formula $\mathsf{len}(|x|, \mathsf{Lang}(x))$ describing all possible lengths of words of $\mathsf{Lang}(x)$. This can be achieved, e.g., by using the lasso-automata construction [4]. Consequently, we extend the formula $\bigwedge_{x \in \mathbb{X}} \mathsf{len}(|x|, \mathsf{Lang}(x))$ with the equality $|x| = |x_1| + \cdots + |x_n|$ for each $(x \mapsto x_1 \cdots x_n) \in \sigma$ and $|x_1| + \cdots + |x_n| = |y_1| + \cdots + |y_m|$ for each equation $x_1 \cdots x_n = y_1 \cdots y_m$ from $\mathcal{E}$ in order to obtain the resulting LIA formula φ_{len}. The LIA formula combines length constraints induced by initial and substituted equations and lengths of the substituted variables. Finally, we check if the LIA formula $\mathcal{L} \wedge \varphi_{\mathsf{len}}$ is satisfiable. Since we work with a stable solution, a LIA model of this formula can be used to construct a string model satisfying the obtained LIA model and, hence, satisfying the formula $\mathcal{E} \wedge \mathcal{R} \wedge \mathcal{L}$.

▶ **Example 2.** An example of obtaining a stable solution using rules of simplified stabilization-based procedure applied on the string constraint $xy = z \wedge w = xx \wedge w \in a^+$ is shown in Figure 1. The relevant part of the LIA formula φ_{len} corresponding to this stable solution is then given as

$$\varphi_{\mathsf{len}} \overset{\text{def}}{\Leftrightarrow} |y| \geq 0 \wedge |x| \geq 1 \wedge |z| \geq 1 \wedge |w| \geq 2 \wedge |v_1| \geq 1 \wedge \\ |x| + |x| = |w| \wedge |z| = |x| + |y| \wedge |x| = |v_1| \wedge |w| = |v_1| + |v_1| \wedge |z| = |v_1| + |y|. \quad \lrcorner$$

5 String-Integer Conversions

We propose an extension of the stabilization-based procedure from Section 4 so that it can handle string constraints of the form $\mathcal{E} \wedge \mathcal{R} \wedge \mathcal{L} \wedge \mathcal{C}$. We know that $\mathcal{E} \wedge \mathcal{R} \wedge \mathcal{L}$ is satisfiable if there is a stable solution (Lang, σ) such that the LIA formula $\mathcal{L} \wedge \varphi_{\mathsf{len}}$ is satisfiable. From a high-level point of view, given a stable solution, our procedure creates a LIA formula $\varphi_{\mathsf{conv}} \overset{\text{def}}{\Leftrightarrow} \bigwedge_{c \in \mathcal{C}} \varphi_c$ encoding conversions from $\mathcal{C}$ so that the string constraint $\mathcal{E} \wedge \mathcal{R} \wedge \mathcal{L} \wedge \mathcal{C}$ is satisfiable iff the LIA formula $\mathcal{L} \wedge \varphi_{\mathsf{len}} \wedge \varphi_{\mathsf{conv}}$ is satisfiable.

Additional constraint generation. In order to simplify the resulting LIA formula for conversions φ_{conv}, we add additional membership constraints to the set of initial regular constraints $\mathcal{R}$. The stabilization-based procedure then works with this modified set of constraints. These additional constraints make the resulting formula smaller and since the stabilization-based procedure is particularly optimized for working with regular constraints, the constraints may significantly speed up the whole procedure. More specifically, we enrich regular constraints with formulae restricting the results of `from_int` and `from_code`; i.e., for each constraint $x = \texttt{from\_int}(k)$ from $\mathcal{C}$ we add the constraint $x \in ((\texttt{'1'} + \cdots + \texttt{'9'})(\texttt{'0'} + \cdots + \texttt{'9'})^*) + \epsilon$, restricting x to a valid representation of a number without leading zeros or ϵ, and for each $x = \texttt{from\_code}(k)$ from $\mathcal{C}$ we add the constraint $x \in \Sigma + \epsilon$ restricting x to be either the symbol whose code point we are computing or ϵ for invalid inputs.

5.1 Handling `to_int`

Let (Lang, σ) be a stable solution of a string constraint without conversions and $k = \texttt{to\_int}(x) \in \mathcal{C}$ be a conversion that we want to encode into a LIA formula $\varphi_{k=\texttt{to\_int}(x)}$. Generally speaking, generating a LIA formula from a regular language $\mathsf{Lang}(x)$ that represents all encoded numbers is not possible, because some non-linear function such as exponentiation is necessary [15]. For example, given the language $\{\texttt{'5'}\}\{\texttt{'0'}\}^*$, the corresponding formula in nonlinear arithmetic would be $\ell = 5 \cdot z \wedge \exists n\colon n \geq 0 \wedge z = 10^n$. For this reason, we assume that the language $\mathsf{Lang}(x)$ is *finite*. This restriction is relatively strong, but in combination with our underapproximations from Section 6, it appears to be permissive enough.

As $\mathsf{Lang}(x)$ is finite, we could easily enumerate all words of $\mathsf{Lang}(x)$ and encode into $\varphi_{k=\texttt{to\_int}(x)}$ that $\texttt{to\_int}(x)$ must be equal to one of these values. This would result in the formula $\bigvee_{w \in \mathsf{Lang}(x)} \texttt{to\_int}(x) = \texttt{to\_int}(w)$, with $\texttt{to\_int}(w)$ being the integer value that the string w represents (or -1 if w is a string that does not encode a number) as defined in Section 3. However, the number of possible words could easily blow up and the resulting formula would be too large. For example, for the language $\{\texttt{'0'}, \ldots, \texttt{'9'}\}^9$, we would need 10^9 disjuncts. We need a more succinct encoding. Moreover, the encoding must allow efficient handling the following two issues:

(i) We need to keep the correspondence between the length of x and the value of $\texttt{to\_int}(x)$. The string constraint may, for instance, look like $|x| \leq 3 \wedge z \geq 1000 \wedge z = \texttt{to\_int}(x)$, which is unsatisfiable due to the relation between the length of the string and the value the string represents.

(ii) There can be other variables whose conversion result depend on x. As an example, assume that $\sigma(x) = z_1 z_2$ and we also have to handle the conversion $m = \texttt{to\_int}(y)$ with $\sigma(y) = z_2 z_3$. This means that x must end with the same string that y starts with. The two formulae $\varphi_{k=\texttt{to\_int}(x)}$ and $\varphi_{m=\texttt{to\_int}(y)}$ obtained from the naive approach, however, do not encode this relation.

Succinct and efficient encoding. To achieve succinctness, we reduce the number of disjuncts by working with sets of strings that represent continuous intervals of numbers. For example, if $\mathsf{Lang}(x) = \{\texttt{'0'}, \ldots, \texttt{'3'}, \texttt{'5'}\}\{\texttt{'0'}, \ldots, \texttt{'9'}\}\{\texttt{'0'}, \ldots, \texttt{'9'}\}$, then we get two intervals of numbers $[0 - 399]$ and $[500 - 599]$, and we can create a significantly smaller LIA formula $\varphi_{k=\texttt{to\_int}(x)} \overset{\text{def}}{\Leftrightarrow} k = \texttt{to\_int}(x) \wedge (0 \leq \texttt{to\_int}(x) \leq 399 \vee 500 \leq \texttt{to\_int}(x) \leq 599)$. The two issues discussed above are then handled as follows:

(i) We relate each interval with the corresponding length.

(ii) We work on the level of substituted variables (according to σ) instead of the original ones. Because σ is flat, we can directly give a formula to define $\texttt{to\_int}$ values of the substituted variables and then use these to define the original ones. Regarding the problematic example above, because we know that σ is flat, it holds that $\sigma(z_1) = z_1$, $\sigma(z_2) = z_2$, and $\sigma(z_3) = z_3$, so we construct formulae for $\texttt{to\_int}(z_1)$, $\texttt{to\_int}(z_2)$, and $\texttt{to\_int}(z_3)$, and then combine them to get the value of the original $\texttt{to\_int}(x)$ and $\texttt{to\_int}(y)$.

Encoding $\texttt{to\_int}(x_i)$. To encode $\texttt{to\_int}(x)$, assuming that $\sigma(x) = x_1 \cdots x_n$, we need to first encode $\texttt{to\_int}(x_i)$ for each x_i. We split this encoding into two formulae: $\varphi^{\mathsf{digit}}_{\texttt{to\_int}(x_i)}$, which encodes all words from $\mathsf{Lang}^{\mathsf{digit}}_{x_i} = \mathsf{Lang}(x_i) \cap \{\texttt{'0'}, \ldots, \texttt{'9'}\}^*$, i.e., words containing only digits; and $\varphi^{\mathsf{nondigit}}_{\texttt{to\_int}(x_i)}$, which encodes all words from $\mathsf{Lang}^{\mathsf{nondigit}}_{x_i} = \mathsf{Lang}(x_i) \setminus \{\texttt{'0'}, \ldots, \texttt{'9'}\}^*$, i.e., words containing at least one non-digit symbol.

To define $\varphi^{\mathsf{digit}}_{\texttt{to\_int}(x_i)}$, we introduce the following notation. Let $L_{x_i} = \{|w| \mid w \in \mathsf{Lang}^{\mathsf{digit}}_{x_i}\}$ denote the set of all possible lengths of words containing only digits in $\mathsf{Lang}(x_i)$. Because we assume that $\mathsf{Lang}(x_i)$ is finite, then also L_{x_i} will be finite. Furthermore, let $I_{x_i}(m)$, for $m \in L_{x_i}$, be the set of pairs $(\mathit{low}, \mathit{high})$ of words of the length m from $\mathsf{Lang}^{\mathsf{digit}}_{x_i}$ that encode the largest possible continuous intervals of numbers of the form $(\texttt{to\_int}(\mathit{low}), \texttt{to\_int}(\mathit{high}))$. For example, for $\mathsf{Lang}(x_i) = \{\epsilon, \texttt{'0'}, \texttt{'1'}\}\{\texttt{'0'}, \texttt{'1'}, \texttt{'2'}\}\{\texttt{'0'}, \ldots, \texttt{'9'}\}$ we have $L_{x_i} = \{2, 3\}$, $I_{x_i}(2) = \{(\texttt{'00'}, \texttt{'29'})\}$, and $I_{x_i}(3) = \{(\texttt{'000'}, \texttt{'029'}), (\texttt{'100'}, \texttt{'129'})\}$. Formally, given an interval encoding $(\mathit{low}, \mathit{high}) \in I_{x_i}(m)$, we have that $|\mathit{low}| = |\mathit{high}| = m$ and $\texttt{to\_int}(\mathit{low}) \leq \texttt{to\_int}(\mathit{high})$. Further, the words with length m that encode numbers $\texttt{to\_int}(\mathit{low}) - 1$ and $\texttt{to\_int}(\mathit{high}) + 1$ are not in $\mathsf{Lang}^{\mathsf{digit}}_{x_i}$ and all words w whose length is m and $\texttt{to\_int}(\mathit{low}) \leq \texttt{to\_int}(w) \leq \texttt{to\_int}(\mathit{high})$ must belong to $\mathsf{Lang}(x_i)$. The formula is then defined as

$$\varphi^{\mathsf{digit}}_{\texttt{to\_int}(x_i)} \overset{\text{def}}{\Leftrightarrow} \bigvee_{\ell \in L_{x_i}} \left(|x_i| = \ell \wedge \bigvee_{(\mathit{low}, \mathit{high}) \in I_{x_i}(\ell)} \texttt{to\_int}(\mathit{low}) \leq \texttt{to\_int}(x_i) \leq \texttt{to\_int}(\mathit{high}) \right)$$

where $\texttt{to\_int}(\mathit{low})$ and $\texttt{to\_int}(\mathit{high})$ denote the integers represented by the string literals low and high respectively. The formula says that for each length ℓ, $\texttt{to\_int}(x_i)$ must be a number encoded by one of the words from the language $\mathsf{Lang}^{\mathsf{digit}}_{x_i}$ whose length is ℓ. Let us note that for $\ell = 0$ there is only one possible interval (ϵ, ϵ) and for this case it holds that $\texttt{to\_int}(x_i) = -1$ (because of the fact that $\texttt{to\_int}(\epsilon) = -1$).

On the other hand, defining formula for words from $\mathsf{Lang}^{\mathsf{nondigit}}_{x_i}$ is simple because for each such word w, we know that $\texttt{to\_int}(w) = -1$. Therefore, we can define it as

$$\varphi^{\mathsf{nondigit}}_{\texttt{to\_int}(x_i)} \overset{\text{def}}{\Leftrightarrow} \mathsf{len}(|x_i|, \mathsf{Lang}^{\mathsf{nondigit}}_{x_i}) \wedge \texttt{to\_int}(x_i) = -1$$

where $\mathsf{len}(|x_i|, \mathsf{Lang}^{\mathsf{nondigit}}_{x_i})$, defined in Section 4, is a LIA formula encoding that $|x_i|$ must be the length of some word from the language $\mathsf{Lang}^{\mathsf{nondigit}}_{x_i}$. Note that because this formula can be created even for an infinite language, we can relax the condition on finiteness of the language $\mathsf{Lang}(x_i)$ to only finiteness of the $\mathsf{Lang}^{\mathsf{digit}}_{x_i}$ part.

Encoding $\texttt{to\_int}(x)$. Having defined the encodings of $\texttt{to\_int}(x_i)$ for all x_i in $\sigma(x) = x_1 \cdots x_n$, we can now use them to define $\varphi_{k=\texttt{to\_int}(x)}$. Again, we split the definition into two formulae, but instead of splitting it based on whether the given word contains a digit, we split it into two cases, based on whether x represents a valid number or not, into formulae $\varphi^{\mathsf{valid}}_{\texttt{to\_int}(x)}$ and $\varphi^{\mathsf{invalid}}_{\texttt{to\_int}(x)}$ respectively.

We start with the definition of a LIA formula $\varphi^{\mathsf{valid}}_{\texttt{to\_int}(x)}$ expressing possible valuations of inputs that represent valid numbers. Notice that for a given combination $(\ell_1, \ldots, \ell_n) \in L_{x_1} \times \cdots \times L_{x_n}$ of lengths, we can easily compute the value $\texttt{to\_int}(x)$ by summing the values of $\texttt{to\_int}(x_i)$ multiplied by the correct power of ten. Therefore, we define $\varphi^{\mathsf{valid}}_{\texttt{to\_int}(x)}$ as the following formula:

$$\varphi^{\mathsf{valid}}_{\texttt{to\_int}(x)} \overset{\text{def}}{\Leftrightarrow} \bigvee_{\substack{\ell_1 \in L_{x_1} \\ \vdots \\ \ell_n \in L_{x_n} \\ \ell_1 + \cdots + \ell_n \neq 0}} \left(\texttt{to\_int}(x) = \sum_{1 \leq i \leq n} \left(\texttt{to\_int}(x_i) \cdot 10^{\ell_{i+1} + \cdots + \ell_n} \cdot \operatorname{sgn} \ell_i \right) \wedge \psi[l_1 \ldots l_n] \right)$$

where sgn is the *sign function* (that sends ℓ_i to 1 if it is positive and to 0 if it is itself 0), the corresponding summand $\texttt{to\_int}(x_i) \cdot 10^{\ell_{i+1} + \cdots + \ell_n}$ has to be equal to 0, and

$$\psi[l_1 \ldots l_n] \overset{\text{def}}{\Leftrightarrow} \bigwedge_{1 \leq i \leq n} \left(|x_i| = \ell_i \wedge (\texttt{to\_int}(x_i) = -1 \Rightarrow \ell_i = 0) \right)$$

connects the length $|x_i|$ with the value of ℓ_i and discards values of x_i that do not represent numbers (except for the empty word ϵ). Note that if some (but not all) variable x_i is an empty word, then $x_1 \cdots x_n$ still represents a valid number. Furthermore, because $|x| = |x_1| + \cdots + |x_n|$ is a part of φ_{len}, the connection between $|x|$ and the value of $\texttt{to\_int}(x)$ can be kept by putting restrictions only on the values $|x_1|, \ldots, |x_n|$.

We now define formula $\varphi^{\mathsf{invalid}}_{\texttt{to\_int}(x)}$ encoding the situation when x does not represent a number and $\texttt{to\_int}(x)$ should be equal to -1:

$$\varphi^{\mathsf{invalid}}_{\texttt{to\_int}(x)} \overset{\text{def}}{\Leftrightarrow} \texttt{to\_int}(x) = -1 \wedge \left(|x| = 0 \vee \bigvee_{1 \leq i \leq n} \left(\texttt{to\_int}(x_i) = -1 \wedge |x_i| \neq 0 \right) \right)$$

The variable x does not represent a number if it is ϵ or if some x_i does not represent a number. However, x_i might not represent a number even in the case that $x_i = \epsilon$, but as already mentioned, x can still be a valid representation of a number, so we discard these cases.

Finally, the resulting formula $\varphi_{k=\texttt{to\_int}(x)}$ is given as

$$\varphi_{k=\texttt{to\_int}(x)} \overset{\text{def}}{\Leftrightarrow} k = \texttt{to\_int}(x) \wedge \left(\varphi^{\mathsf{valid}}_{\texttt{to\_int}(x)} \vee \varphi^{\mathsf{invalid}}_{\texttt{to\_int}(x)} \right) \wedge \bigwedge_{1 \leq i \leq n} \left(\varphi^{\mathsf{digit}}_{\texttt{to\_int}(x_i)} \vee \varphi^{\mathsf{nondigit}}_{\texttt{to\_int}(x_i)} \right).$$

We note that in the case $\varphi^{\mathsf{valid}}_{\texttt{to\_int}(x)}$ is true, its conjunct $\psi[l_1 \ldots l_n]$ forces $\varphi^{\mathsf{nondigit}}_{\texttt{to\_int}(x_i)}$ to be false for all x_i.

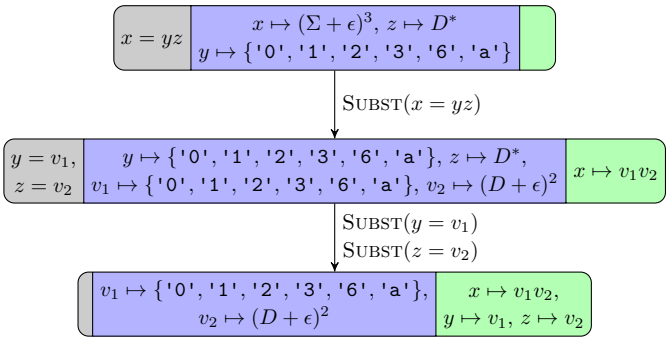


Figure 2 A part of the proof graph generated by the rules of the simplified stabilization procedure for the string constraint $x = yz \wedge x \in (\Sigma + \epsilon)^3 \wedge y \in \{\texttt{'0'}, \texttt{'1'}, \texttt{'2'}, \texttt{'3'}, \texttt{'6'}, \texttt{'a'}\} \wedge z \in D^*$, where $D = \{\texttt{'0'}, \ldots, \texttt{'9'}\}$.

▶ **Example 3.** Let $D = \{\texttt{'0'}, \ldots, \texttt{'9'}\}$ be the set of digits and

$$\psi \overset{\text{def}}{\Leftrightarrow} x = yz \wedge x \in (\Sigma + \epsilon)^3 \wedge y \in \{\texttt{'0'}, \texttt{'1'}, \texttt{'2'}, \texttt{'3'}, \texttt{'6'}, \texttt{'a'}\} \wedge z \in D^* \wedge i = \texttt{to\_int}(x)$$

a string constraint that we want to solve. Figure 2 shows a run of the stabilization procedure on this string constraint (without the `to_int` conversion) that results in the stable solution (Lang, σ), where $\mathsf{Lang}(v_1) = \{\texttt{'0'}, \texttt{'1'}, \texttt{'2'}, \texttt{'3'}, \texttt{'6'}, \texttt{'a'}\}$, $\mathsf{Lang}(v_2) = (D + \epsilon)^2$, $\sigma(x) = v_1 v_2$, $\sigma(y) = v_1$, and $\sigma(z) = v_2$ (mappings of other variables in Lang and σ are not relevant). The relevant part of the LIA formula φ_{len} corresponding to this stable solution is then given as

$$\varphi_{\mathsf{len}} \overset{\text{def}}{\Leftrightarrow} (|v_1| = 1 \vee |v_1| = 2) \wedge 0 \leq |v_2| \leq 2 \wedge |x| = |v_1| + |v_2| \wedge |y| = |v_1| \wedge |z| = |v_2|.$$

We need to create the formula $\varphi_{i=\texttt{to\_int}(x)}$ and because $\sigma(x) = v_1 v_2$, we need to first create the formulae $\varphi^{\mathsf{digit}}_{\texttt{to\_int}(v_1)}$, $\varphi^{\mathsf{nondigit}}_{\texttt{to\_int}(v_1)}$, $\varphi^{\mathsf{digit}}_{\texttt{to\_int}(v_2)}$, and $\varphi^{\mathsf{nondigit}}_{\texttt{to\_int}(v_2)}$. Starting with v_1, we have $\mathsf{Lang}^{\mathsf{digit}}_{v_1} = \{\texttt{'0'}, \texttt{'1'}, \texttt{'2'}, \texttt{'3'}, \texttt{'6'}\}$, $\mathsf{Lang}^{\mathsf{nondigit}}_{v_1} = \{\texttt{'a'}\}$, and $L_{v_1} = \{1\}$. Therefore, we get

$$\varphi^{\mathsf{digit}}_{\texttt{to\_int}(v_1)} \overset{\text{def}}{\Leftrightarrow} |v_1| = 1 \wedge (0 \leq \texttt{to\_int}(v_1) \leq 3 \vee \texttt{to\_int}(v_1) = 6) \text{ and}$$
$$\varphi^{\mathsf{nondigit}}_{\texttt{to\_int}(v_1)} \overset{\text{def}}{\Leftrightarrow} |v_1| = 1 \wedge \texttt{to\_int}(v_1) = -1.$$

For v_2, we have $\mathsf{Lang}^{\mathsf{digit}}_{v_2} = (D + \epsilon)^2$, $\mathsf{Lang}^{\mathsf{nondigit}}_{v_2} = \emptyset$, and $L_{v_2} = \{0, 1, 2\}$. We obtain

$$\varphi^{\mathsf{digit}}_{\texttt{to\_int}(v_2)} \overset{\text{def}}{\Leftrightarrow} (|v_2| = 0 \wedge \texttt{to\_int}(v_2) = -1) \vee (|v_2| = 1 \wedge 0 \leq \texttt{to\_int}(v_2) \leq 9) \vee (|v_2| = 2 \wedge 0 \leq \texttt{to\_int}(v_2) \leq 99)$$

and, since $\mathsf{Lang}^{\mathsf{nondigit}}_{v_2}$ is empty, $\varphi^{\mathsf{nondigit}}_{\texttt{to\_int}(v_2)}$ is false. The formula $\varphi^{\mathsf{valid}}_{\texttt{to\_int}(x)}$ is then given as

$$\begin{aligned}\varphi^{\mathsf{valid}}_{\texttt{to\_int}(x)} \overset{\text{def}}{\Leftrightarrow}\ & \big(\texttt{to\_int}(x) = \texttt{to\_int}(v_1) \wedge |v_1| = 1 \wedge |v_2| = 0 \wedge \texttt{to\_int}(v_1) \neq -1\big) \\ \vee\ & \big(\texttt{to\_int}(x) = \texttt{to\_int}(v_1) \cdot 10 + \texttt{to\_int}(v_2) \wedge |v_1| = 1 \\ & \qquad \wedge |v_2| = 1 \wedge \texttt{to\_int}(v_1) \neq -1 \wedge \texttt{to\_int}(v_2) \neq -1\big) \\ \vee\ & \big(\texttt{to\_int}(x) = \texttt{to\_int}(v_1) \cdot 100 + \texttt{to\_int}(v_2) \wedge |v_1| = 1 \\ & \qquad \wedge |v_2| = 2 \wedge \texttt{to\_int}(v_1) \neq -1 \wedge \texttt{to\_int}(v_2) \neq -1\big)\end{aligned}$$

and $\varphi^{\mathsf{invalid}}_{\texttt{to\_int}(x)}$ as

$$\varphi^{\mathsf{invalid}}_{\texttt{to\_int}(x)} \overset{\text{def}}{\Leftrightarrow} \texttt{to\_int}(x) = -1 \wedge \\ \big(|x| = 0 \vee (\texttt{to\_int}(v_1) = -1 \wedge |v_1| \neq 0) \vee (\texttt{to\_int}(v_2) = -1 \wedge |v_2| \neq 0)\big).$$

The final formula $\varphi_{i=\texttt{to\_int}(x)}$ is then

$$\varphi_{i=\texttt{to\_int}(x)} \overset{\text{def}}{\Leftrightarrow} i = \texttt{to\_int}(x) \wedge \left(\varphi^{\mathsf{valid}}_{\texttt{to\_int}(x)} \vee \varphi^{\mathsf{invalid}}_{\texttt{to\_int}(x)}\right) \wedge \left(\varphi^{\mathsf{digit}}_{\texttt{to\_int}(v_1)} \vee \varphi^{\mathsf{nondigit}}_{\texttt{to\_int}(v_1)}\right) \wedge \varphi^{\mathsf{digit}}_{\texttt{to\_int}(v_2)}$$

which is the only conjunct of φ_{conv}. ⌟

5.2 Handling `from_int`

We briefly describe the handling of `from_int`. Assume that $x = \texttt{from\_int}(k)$ is the conversion from $\mathcal{C}$ that we want to encode and $\sigma(x) = x_1 \cdots x_n$. We proceed similarly to the `to_int` case, but instead of encoding the possible values of the result x, we will instead restrict the argument k to values that yield the given possible values of the result x. The formula $\varphi^{\mathsf{valid}}_{x=\texttt{from\_int}(k)}$ expressing valid conversions is the same as $\varphi^{\mathsf{valid}}_{\texttt{to\_int}(x)}$ after replacing $\texttt{to\_int}(x)$ with k:

$$\varphi^{\mathsf{valid}}_{x=\texttt{from\_int}(k)} \overset{\text{def}}{\Leftrightarrow} \bigvee_{\substack{\ell_1 \in L_{x_1} \\ \vdots \\ \ell_n \in L_{x_n} \\ \ell_1 + \cdots + \ell_n \neq 0}} \left(k = \sum_{1 \leq i \leq n} \left(\texttt{to\_int}(x_i) \cdot 10^{\ell_{i+1} + \cdots + \ell_n} \cdot \operatorname{sgn} \ell_i\right) \wedge \psi[l_1 \ldots l_n]\right).$$

Furthermore, $\texttt{from\_int}(k)$ always returns the string encoding a *non-negative* number k, except for the case when $k < 0$. In this case, it returns ϵ, therefore, the formula $\varphi^{\mathsf{invalid}}_{x=\texttt{from\_int}(k)}$ is defined as

$$\varphi^{\mathsf{invalid}}_{x=\texttt{from\_int}(k)} \overset{\text{def}}{\Leftrightarrow} k < 0 \wedge |x| = 0.$$

The resulting formula $\varphi_{x=\texttt{from\_int}(k)}$ is then given as

$$\varphi_{x=\texttt{from\_int}(k)} \overset{\text{def}}{\Leftrightarrow} \left(\varphi^{\mathsf{valid}}_{x=\texttt{from\_int}(k)} \vee \varphi^{\mathsf{invalid}}_{x=\texttt{from\_int}(k)}\right) \wedge \bigwedge_{1 \leq i \leq n} \left(\varphi^{\mathsf{digit}}_{\texttt{to\_int}(x_i)} \vee \varphi^{\mathsf{nondigit}}_{\texttt{to\_int}(x_i)}\right).$$

▶ **Example 4.** Let us take the string constraint from Example 3 extended with $i \leq j \wedge z = \texttt{from\_int}(j)$. As we added only LIA and conversion constraints, the stable solution (Lang, σ) from Example 3 stays the same. We know that $\sigma(z) = v_2$, $\mathsf{Lang}(v_2) = (D + \epsilon)^2$, and $L_{v_2} = \{0, 1, 2\}$. We also already have $\varphi^{\mathsf{digit}}_{\texttt{to\_int}(v_2)}$ from Example 3 ($\varphi^{\mathsf{nondigit}}_{\texttt{to\_int}(v_2)}$ is unsatisfiable), therefore, in order to encode $\varphi_{z=\texttt{from\_int}(j)}$, we only need

$$\varphi^{\mathsf{valid}}_{z=\texttt{from\_int}(j)} \overset{\text{def}}{\Leftrightarrow} (j = \texttt{to\_int}(v_2) \wedge |v_2| = 1 \wedge \texttt{to\_int}(v_2) \neq -1) \vee \\ (j = \texttt{to\_int}(v_2) \wedge |v_2| = 2 \wedge \texttt{to\_int}(v_2) \neq -1) \text{ and}$$

$$\varphi^{\mathsf{invalid}}_{z=\texttt{from\_int}(j)} \overset{\text{def}}{\Leftrightarrow} j < 0 \wedge |z| = 0.$$

We can then add

$$\varphi_{z=\texttt{from\_int}(j)} \overset{\text{def}}{\Leftrightarrow} \left(\varphi^{\mathsf{valid}}_{z=\texttt{from\_int}(i)} \vee \varphi^{\mathsf{invalid}}_{z=\texttt{from\_int}(i)}\right) \wedge \varphi^{\mathsf{digit}}_{\texttt{to\_int}(v_2)}$$

to φ_{conv}. ⌟

5.3 Handling `to_code`

The semantic of $\texttt{to\_code}(x)$ is simple (see Section 3): if $|x| = 1$, then $\texttt{to\_code}(x)$ represents the code point of the symbol x, otherwise the result is -1. We want to encode this behavior in the formula $\varphi_{k=\texttt{to\_code}(x)}$. Assuming that $\sigma(x) = x_1 \cdots x_n$, we will first, for each x_i, encode the value $\texttt{to\_code}(x_i)$ in a formula $\varphi_{\texttt{to\_code}(x_i)}$, so that we relate the substituted variables between multiple `to_code` conversions (similarly as in `to_int` case). We could take a naive approach and enumerate all possible symbols of x_i (there is only a finite number of possible symbols, so this method is complete). However, such a naive approach could easily blow-up.

In order to overcome this blow-up problem, we can notice that, despite the high number of possible symbols, the input string formula usually uses only a small subset of them (usually a subset of ASCII symbols). We therefore restrict the alphabet Σ only to these symbols, all digit symbols, and one special symbol δ that represents all *unused*[4] *non-digit* symbols (if there are any). Using only δ for all unused symbols has no impact on the stabilization-based procedure from Section 4, as it handles only a conjunction of positive equations with regular and length constraints, where unused symbols "behave in the same way". However, digit symbols are important for `to_int`/`from_int` conversions. We therefore keep all of them in Σ and δ then represents all unused symbols that are not digits.

We then encode the fact that either $\texttt{to\_code}(x_i)$ is a code point of the symbol x_i, or, if x_i is not a single symbol, $\texttt{to\_code}(x_i) = -1$:

$$\varphi_{\texttt{to\_code}(x_i)} \overset{\text{def}}{\Leftrightarrow} \left(\left(|x_i| = 1 \wedge \bigvee_{a \in \mathsf{Lang}(x_i) \cap \Sigma} \psi(a) \right) \vee \left(|x_i| \neq 1 \wedge \texttt{to\_code}(x_i) = -1 \right) \right)$$

where $\psi(a)$ encodes that $\texttt{to\_code}(x_i)$ is equal to the code-point of the symbol a as follows:

- For a normal symbol $a \neq \delta$, we have $\psi(a) \overset{\text{def}}{\Leftrightarrow} \texttt{to\_code}(x_i) = \texttt{to\_code}(a)$ (recall that $\texttt{to\_code}(a)$ for $a \in \Sigma \setminus \{\delta\}$ denotes the numeric value of the code point of the symbol a).
- For δ, we have to encode the fact that $\texttt{to\_code}(\delta)$ can acquire any possible code point of unused symbols. Therefore, given that $\mathsf{maxchar}$ denotes the largest possible code point[5],

$$\psi(\delta) \overset{\text{def}}{\Leftrightarrow} 0 \leq \texttt{to\_code}(x_i) \leq \mathsf{maxchar} \wedge \bigwedge_{b \in \mathsf{Lang}(x_i) \cap (\Sigma \setminus \{\delta\})} \texttt{to\_code}(x_i) \neq \texttt{to\_code}(b)$$

We can then define $\varphi_{\texttt{to\_code}(x)}$ as

$$\varphi_{\texttt{to\_code}(x)} \overset{\text{def}}{\Leftrightarrow} \left(\left(|x| = 1 \wedge \texttt{to\_code}(x) \neq -1 \wedge \bigvee_{1 \leq i \leq n} \texttt{to\_code}(x) = \texttt{to\_code}(x_i) \right) \vee \left(|x| \neq 1 \wedge \texttt{to\_code}(x) = -1 \right) \right)$$

relating the value x with the substituted variables x_i. The second part of the formula handles the case when x is not a symbol. For the case that x is a symbol, there must be exactly one x_i that is also the (same) symbol, and all other x_j, for $i \neq j$, must be empty strings. For each such x_j, we have $\texttt{to\_code}(x_j) = -1$, so by requiring $\texttt{to\_code}(x) \neq -1$, we force the equality $\texttt{to\_code}(x) = \texttt{to\_code}(x_i)$. The resulting formula is then

$$\varphi_{k=\texttt{to\_code}(x)} \overset{\text{def}}{\Leftrightarrow} k = \texttt{to\_code}(x) \wedge \varphi_{\texttt{to\_code}(x)} \wedge \bigwedge_{1 \leq i \leq n} \varphi_{\texttt{to\_code}(x_i)}.$$

---

[4] Unused here means that they are not *explicitly* used, so for example in the formula $x \in \Sigma \wedge y \in \{a, b\}$, the explicitly used symbols are only a and b, and δ will represent all other symbols. We then get a formula with three symbols: $x \in \{a, b, \delta\} \wedge y \in \{a, b\}$.

[5] According to the SMT standard [10], it is the number 196,607.

Connecting string and code conversions. Finally, we need to address the problem of relating `to_code` and `to_int`. For example, we could have both conversions $\texttt{to\_int}(x)$ and $\texttt{to\_code}(y)$ with $\sigma(x) = z_1 z_2$ and $\sigma(y) = z_2 z_3$. Right now, $\texttt{to\_int}(z_2)$ and $\texttt{to\_code}(z_2)$ can have some valid values, which are, however, not related. For example, if $\mathsf{Lang}(z_2) = \{\texttt{'4'}, \texttt{'5'}\}$, then we could end up with the situation where $\texttt{to\_int}(z_2) = 5$ but $\texttt{to\_code}(z_2) = 52$, which is the code point of `'4'`.

Therefore, for each variable x for which both $\varphi_{\texttt{to\_int}(x)}$ and $\varphi_{\texttt{to\_code}(x)}$ are defined, we create the following formula

$$|x| \neq -1 \vee (\texttt{to\_int}(x) = -1 \wedge \neg(48 \leq \texttt{to\_code}(x) \leq 57)) \vee (\texttt{to\_code}(x) = \texttt{to\_int}(x) + 48),$$

which we add to the conversion formula φ_{conv}. This formula represents three (non-necessarily disjoint) cases: (i) the string x is not a symbol, which means that $\texttt{to\_code}(x) = -1$ and $\texttt{to\_int}(x)$ does not depend on the value of $\texttt{to\_code}(x)$; (ii) x is not a digit, therefore $\texttt{to\_code}(x)$ cannot lie in the interval $[48, 57]$, which corresponds to digits `'0'` through `'9'`; and (iii) x is a digit, so $\texttt{to\_code}(x)$ must be its code point.

▶ **Example 5.** We further extend the string constraint from Example 4 by the constraint $|x| = \texttt{to\_code}(y)$. Again, we only added a conversion to the formula, therefore the stable solution (Lang, σ) from Example 3 stays the same, and we only need to encode the formula $\varphi_{|x|=\texttt{to\_code}(y)}$ with $\sigma(y) = v_1$ and $\mathsf{Lang}(v_1) = \{\texttt{'0'}, \texttt{'1'}, \texttt{'2'}, \texttt{'3'}, \texttt{'6'}, \texttt{'a'}\}$. We therefore get the following formulae:

$$\begin{aligned}
\varphi_{\texttt{to_code}(v_1)} \overset{\text{def}}{\Leftrightarrow} \big(|v_1| = 1 \wedge (&\texttt{to_code}(v_1) = 48 \vee \\
&\texttt{to_code}(v_1) = 49 \vee \\
&\texttt{to_code}(v_1) = 50 \vee \\
&\texttt{to_code}(v_1) = 51 \vee \\
&\texttt{to_code}(v_1) = 54 \vee \\
&\texttt{to_code}(v_1) = 97)\big) \vee \\
\big(|v_1| \neq 1 \wedge \texttt{to_code}(v_1) = -1\big)&
\end{aligned}$$

$$\begin{aligned}
\varphi_{\texttt{to_code}(y)} \overset{\text{def}}{\Leftrightarrow} &\big(|y| = 1 \wedge \texttt{to_code}(y) \neq -1 \wedge \texttt{to_code}(y) = \texttt{to_code}(v_1)\big) \vee \\
&\big(|y| \neq 1 \wedge \texttt{to_code}(y) = -1\big)
\end{aligned}$$

$$\varphi_{|x|=\texttt{to\_code}(y)} \overset{\text{def}}{\Leftrightarrow} |x| = \texttt{to\_code}(y) \wedge \varphi_{\texttt{to\_code}(y)} \wedge \varphi_{\texttt{to\_code}(x_i)}$$

Note that in Example 3 we defined $\texttt{to\_int}(v_1)$ and we have also now defined $\texttt{to\_code}(v_1)$, therefore we also need to add the formula connecting these two values together as explained in the previous paragraph. All in all, the final conversion formula φ_{conv} is then defined as

$$\begin{aligned}
\varphi_{\mathsf{conv}} \overset{\text{def}}{\Leftrightarrow} &\big(|v_1| \neq -1 \vee (\texttt{to_int}(v_1) = -1 \wedge \neg(48 \leq \texttt{to_code}(v_1) \leq 57)) \vee \\
&(\texttt{to_code}(v_1) = \texttt{to_int}(v_1) + 48)\big) \wedge \\
&\varphi_{i=\texttt{to_int}(x)} \wedge \varphi_{z=\texttt{from_int}(j)} \wedge \varphi_{|x|=\texttt{to_code}(y)}.
\end{aligned}$$

5.4 Handling `from_code`

Similarly to `from_int`, which is handled using `to_int`, `from_code` is handled using `to_code`. Given $x = \texttt{from\_code}(k)$ and $\sigma(x) = x_1 \cdots x_n$, the formula $\varphi_{x=\texttt{from\_code}(k)}$ is defined as

$$\begin{aligned}
\varphi_{x=\texttt{from_code}(k)} \overset{\text{def}}{\Leftrightarrow} \Bigg(\Big(|x| = 1 \wedge k \neq -1 \wedge \bigvee_{1 \leq i \leq n} k = \texttt{to_code}(x_i)\Big) \vee \\
\Big(|x| = 0 \wedge \neg(0 \leq k \leq \mathsf{maxchar})\Big)\Bigg) \wedge \bigwedge_{1 \leq i \leq n} \varphi_{\texttt{to_code}(x_i)}
\end{aligned}$$

The only two differences from `to_code` are the following:

1. we are restricting the values of the argument k (instead of computing $\texttt{to\_code}(x)$) and
2. for the case that x is not a symbol (which must mean that $x = \epsilon$, as $\texttt{from\_code}(k)$ returns either a symbol for a valid code point or an empty string, hence the condition $|x| = 0$), the argument can be any integer that is an invalid code point.

5.5 Handling word disequations through `to_code`

The stabilization-based procedure from Section 4 assumes that $\mathcal{E}$ does not contain disequations. As shown in [18], every disequation $s \neq t$ can be encoded into a combination of equations and length constraints as

$$\varphi_{s\neq t} \overset{\text{def}}{\Leftrightarrow} |s| \neq |t| \vee \left(s = x_1a_1y_1 \wedge t = x_2a_2y_2 \wedge |x_1| = |x_2| \wedge a_1 \in \Sigma \wedge a_2 \in \Sigma \wedge \overbrace{a_1 \neq a_2}^{\mathsf{dist}(a_1,a_2)}\right)$$

where x_1, x_2, y_1, y_2, a_1, and a_2 are fresh variables and the disequation $a_1 \neq a_2$ between symbols can be encoded into a LIA formula $\mathsf{dist}(a_1, a_2)$ after the procedure returns a stable solution. Furthermore, if the original string constraint without disequations is chain-free, then it stays chain-free even if we add $\varphi_{s\neq t}$ to it. However, the transformation of $a_1 \neq a_2$ to LIA formula $\mathsf{dist}(a_1, a_2)$ from [18] is incompatible with the way we construct φ_{conv}. We can easily solve this by replacing $a_1 \neq a_2$ in $\varphi_{s\neq t}$ with $\texttt{to\_code}(a_1) \neq \texttt{to\_code}(a_2)$. Since our procedure can handle any `to_code` conversion, this extends one of the richest decidable fragments of string constraints, chain-free constraints with lengths and arbitrary disequations [18], by code-point conversions.

6 Implementation and Optimizations

We implemented the proposed technique into the string solver Z3-Noodler [34], which implements an optimized version of the stabilization-based procedure from Section 4 within the DPLL(T)-based SMT solver Z3. The LIA formula φ_{conv} for string-integer conversions is constructed after the procedure finds a stable solution. The formula φ_{conv} is appended to the length formula φ_{len} and the result is checked for satisfiability. On top of the proposed technique, we further introduced several optimizations, described below.

Underapproximation of `to_int`. We mentioned in Section 5 that the conversion $\texttt{to\_int}(x)$ expects the language $\mathsf{Lang}_x^{\mathsf{digit}}$, the subset of words from $\mathsf{Lang}(x)$ containing only digits, to be finite. Although this restriction usually holds in the benchmarks, we implemented an *underapproximation* to at least partially handle the case of infinite languages. In particular, if a variable occurring in a conversion has an infinite language after stabilization, we restrict the language to strings up to some fixed length. To be more concrete, if $\mathsf{Lang}_x^{\mathsf{digit}}$ of a variable x is an infinite language, we restrict it to $\mathsf{Lang}_x^{\mathsf{digit}} \cap (\Sigma+\epsilon)^m$ where m is the underapproximation parameter (we use $m = 5$ in the experiments). This approach is sound for the SAT case.

Supporting constraints. In order to keep the variable languages as precise as possible, we generate additional constraints (axioms) that are used to enrich the initial string constraint. One such kind of constraints is generated for conversions of the type $x = \texttt{from\_int}(k)$, which always result in x being assigned an infinite language (unless it is restricted by some other regular constraint). Therefore, we generate an additional constraint $k < 10^m \implies x \in (\Sigma + \epsilon)^m$ where m is the same underapproximation parameter as in the previous paragraph, which is then added to the initial constraint. Furthermore, we infer regular constraints from LIA formulae. For example, if we have $\texttt{to\_int}(x) = k$, with $k \in \mathbb{Z}$, we generate $x \in \texttt{'0'}^* \cdot \texttt{from\_int}(k)$. Moreover, if we have $|x| \leq k$, we can set $x \in (\Sigma + \epsilon)^k$.

Table 1 Results of experiments on all benchmark sets. For each benchmark set we give the number of solved instances, the number of unknowns, and the number of instances where the particular solver runs out of resources (timeout or out of memory). Moreover, we measure each value w.r.t. all instances in the benchmark set (*all*) as well as only to instances containing at least one conversion function (*conv*). The number of particular instances is given in the benchmark header.

	FullStrInt (16,968 | 16,130)						StringFuzz (11,618 | 1,608)						StrSmallRw (1,880 | 80)						Σ	
	solved		***unknown***		***OOR***		***solved***		***unknown***		***OOR***		***solved***		***unknown***		***OOR***		***solved***	
	all	*conv*	*all*	*conv*	*all*	*conv*	*all*	*conv*	*all*	*conv*	*all*	*conv*	*all*	*conv*	*all*	*conv*	*all*	*conv*	*all*	*conv*
Z3-Noodler	16,704	15,872	126	126	138	132	**11,616**	**1,606**	2	2	**0**	**0**	1,743	73	100	1	37	6	**30,063**	17,551
cvc5	**16,963**	**16,125**	**0**	**0**	5	5	10,915	1,579	**0**	**0**	703	29	**1,861**	**78**	2	2	**17**	**0**	29,739	**17,782**
Z3	16,729	15,896	**0**	**0**	239	234	11,081	1,565	**0**	**0**	537	43	1,821	**78**	**0**	**0**	59	2	29,631	17,539
OSTRICH	15,909	15,109	**0**	**0**	1,059	1,021	11,400	1,558	**0**	**0**	217	50	1,709	69	**0**	**0**	171	11	29,018	16,736
Z3-Noodler^{pr}	11,665	10,857	5,299	5,273	**4**	**0**	10,050	41	1,568	1,567	**0**	**0**	1,615	62	210	18	55	**0**	23,330	10,960

Interval computation. The solver Z3-Noodler internally represents each $\mathsf{Lang}(x)$ using NFAs. In order to efficiently create the set of intervals $I_{x_i}(n)$ for encoding $\texttt{to\_int}(x_i)$ (cf. Section 5.1), we minimize the automaton for $\mathsf{Lang}(x_i)$, getting deterministic automaton with a finite language, i.e., from each state there is at most one transition per symbol and there are no loops. It is therefore easy to create intervals of digits going from each state, and then connect these short intervals into longer ones.

7 Experiments

Used tools and environment. We compared Z3-Noodler extended with the support of string-integer conversions (version 1.1.0) with the other state-of-the-art tools: cvc5 (version 1.1.2), Z3 (version 4.13.0), and OSTRICH (the latest commit `5dd2e10`). We also add comparison with the previous version 1.0.0 of Z3-Noodler (denoted as Z3-Noodler^{pr}). We do not include comparison with Z3-Trau and Z3str4 as they give incorrect results on some instances. The experiments were executed on a workstation with an AMD Ryzen 5 5600G CPU @ 3.8 GHz with 100 GiB of RAM running Ubuntu 22.04.4. The timeout was set to 120 s, memory limit was set to 8 GiB.

Benchmarks. For the experimental evaluation we selected all benchmark sets from SMT-LIB [9] (category `QF_SLIA`) containing string-integer conversions. Concretely, we took the FullStrInt (16,968 formulae), StringFuzz (11,618 formulae), and StrSmallRw (1,880 formulae) benchmark sets, which were also used in SMT-COMP'23. Except string-integer conversions, these benchmarks make heavy use of string (dis)equations and other string predicates and functions (e.g., `substr`, `indexof`, `at`, ...) combined with length and regular constraints.

Results. The results summarizing the number of solved instances are given in Table 1 and the average running times are shown in Table 2. The scatter plots comparing the running times of Z3-Noodler and the other tools are given in Figure 3.

Regarding the FullStrInt benchmark set, Z3-Noodler has 138 timeouts (out of which 132 were on formulae with conversions) and 126 unknowns. The number of solved instances is comparable to Z3, but smaller compared to cvc5 (cvc5 solves all instances except 5). On the other hand, Z3-Noodler solved 795 instances more than OSTRICH. Concerning the running time, Z3-Noodler has the lowest average time followed by cvc5 (not counting Z3-Noodler^{pr} as it has almost 5,300 unknowns on FullStrInt). The inspection of timeouts of Z3-Noodler on FullStrInt revealed that the bottleneck on these examples is not the handling of string-integer conversions but handling of a complex combination of string predicates and functions (`substr`, `indexof`, ...) in the stabilization-based procedure. We believe that these

Table 2 Average running times and standard deviation (in seconds) of *solved* instances.

	FullStrInt		StringFuzz		StrSmallRw	
	avg	***std***	***avg***	***std***	***avg***	***std***
Z3-Noodler	0.19	1.26	**0.03**	0.22	0.14	2.41
cvc5	0.30	1.42	2.81	12.55	**0.02**	0.56
Z3	1.21	6.90	4.14	15.39	0.12	2.86
OSTRICH	11.86	14.85	4.79	10.00	4.47	6.61
Z3-Noodler^{pr}	**0.02**	0.05	**0.03**	0.08	0.75	6.56

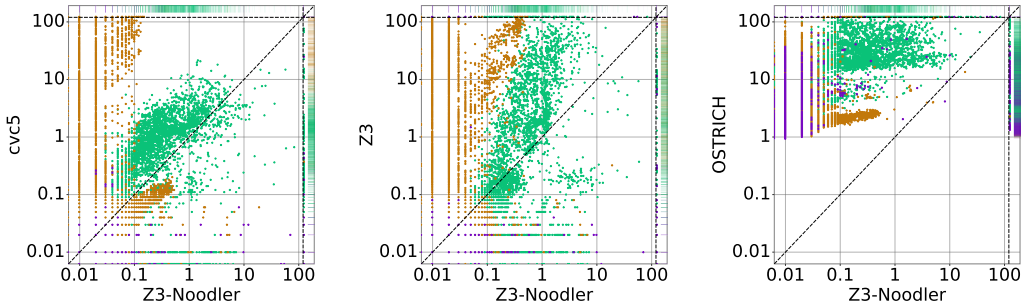
instances could be solved by a tailored preprocessing of input formulae and strengthening axioms for special cases of string functions and predicates. The unknowns of Z3-Noodler are caused by the inconclusive result in the case of underapproximation. It is evident from the table that Z3-Noodler can solve significantly more instances than Z3-Noodler^{pr}. Note that although Z3-Noodler^{pr} has no support for string-integer conversions, it can solve some instances, for example those containing unsatisfiable LIA parts or those whose satisfiability can be established without considering the string-integer conversions.

In the StringFuzz benchmark set, Z3-Noodler solved more instances than any other tool. In particular, Z3-Noodler has only 2 unknowns (caused by the underapproximation), which is significantly better than the runner-up OSTRICH. Concerning the running time on this benchmark set, Z3-Noodler has also the lowest average time.

The last considered benchmark set StrSmallRw is not very interesting. It contains only 80 formulae with conversions and from these, already 62 are solved by Z3-Noodler^{pr}, i.e., they can be solved without looking at conversions. If we take into account the whole benchmark, Z3-Noodler has a large amount of unknowns, which are caused by a limited support of the ¬`contains` predicate. The average running time of Z3-Noodler is comparable to Z3, but higher than the average time of cvc5, which is the fastest tool on this benchmark.

From the summary statistics in Table 1 it is evident that the proposed procedure and its integration with the stabilization-based procedure has a significant effect on the number of solved instances. The previous version Z3-Noodler^{pr} solved 6,734 instances less than Z3-Noodler. If we look at the number of solved cases from the perspective of all instances, Z3-Noodler solved the most formulae compared to any other tool. If we restrict our attention only to formulae with conversions, Z3-Noodler solves 231 instances less than cvc5, but more instances than Z3 and OSTRICH. From Figure 3 and the average running times in Table 2, it is obvious that Z3-Noodler can be (sometimes significantly) faster than other state-of-the-art tools pointing out to the efficiency of the proposed combined approach.

Discussion. The experimental evaluation shows that our proposed approach combining the stabilization-based procedure and the translation of the conversions into a LIA formula can (sometimes significantly) improve the scalability of solving complex string constraints containing string-integer conversions. Although the generated LIA formula might be large in the worst-case, based on our experiments, this worst case is avoided in real-world benchmarks (based on inspecting a couple of instances, the generated LIA formulae contained up to hundreds of simple atoms). The experiments also revealed that the highest impact on the LIA formula size was the interval separation (Section 5.1); without this feature the LIA formula often exploded. Furthermore, based on our experience, the generated LIA formula is easily solvable by Z3's LIA solver. The current bottleneck of the solver is not handling the string-integer conversions but handling the complex combination of string predicates and functions (as discussed at the results of FullStrInt).



(a) Z3-Noodler vs cvc5. **(b)** Z3-Noodler vs Z3. **(c)** Z3-Noodler vs OSTRICH.

Figure 3 Scatter plots comparing Z3-Noodler with cvc5, Z3, and OSTRICH. Times are in seconds, axes are logarithmic. The dashed lines represent unknowns and timeouts/out-of-memory. Colours distinguish benchmark sets: • FullStrInt, • StringFuzz, and • StrSmallRw.

8 Conclusion

We have proposed an extension of the stabilization-based procedure with the handling of string-integer conversions. Based on a stable solution, the technique encodes the conversions into LIA formulae. We have proposed a series of optimizations in order to avoid a blow-up of the LIA formulae and implemented the approach in the tool Z3-Noodler. Our experimental evaluation on established benchmarks shows that our technique outperforms other state-of-the-art tools on many instances.

In the future, we plan to extend the precise handling of conversions from finite languages to languages with limited forms of iteration. Another direction could be a dynamic analysis of the underapproximation parameter and adjusting the parameter according to the input formula characteristics.

References

1 Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Bui Phi Diep, Julian Dolby, Petr Janku, Hsin-Hung Lin, Lukás Holík, and Wei-Cheng Wu. Efficient handling of string-number conversion. In Alastair F. Donaldson and Emina Torlak, editors, *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 943–957. ACM, 2020. doi:10.1145/3385412.3386034.

2 Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Bui Phi Diep, Lukás Holík, Ahmed Rezine, and Philipp Rümmer. Flatten and conquer: a framework for efficient analysis of string constraints. In Albert Cohen and Martin T. Vechev, editors, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 602–617. ACM, 2017. doi:10.1145/3062341.3062384.

3 Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Bui Phi Diep, Lukás Holík, Ahmed Rezine, and Philipp Rümmer. Trau: SMT solver for string constraints. In Nikolaj S. Bjørner and Arie Gurfinkel, editors, *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018*, pages 1–5. IEEE, 2018. doi:10.23919/FMCAD.2018.8602997.

4 Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Lukás Holík, Ahmed Rezine, Philipp Rümmer, and Jari Stenman. String constraints for verification. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 150–166. Springer, 2014. doi:10.1007/978-3-319-08867-9_10.

5 Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Lukáš Holík, Ahmed Rezine, Philipp Rümmer, and Jari Stenman. Norn: An SMT solver for string constraints. In *CAV'15*, volume 9206 of *LNCS*, pages 462–469. Springer, 2015.

6 Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bui Phi Diep, Lukás Holík, and Petr Janků. Chain-free string constraints. In Yu-Fang Chen, Chih-Hong Cheng, and Javier Esparza, editors, *Automated Technology for Verification and Analysis - 17th International Symposium, ATVA 2019, Taipei, Taiwan, October 28-31, 2019, Proceedings*, volume 11781 of *Lecture Notes in Computer Science*, pages 277–293. Springer, 2019. doi:10.1007/978-3-030-31784-3_16.

7 Leonardo Alt, Martin Blicha, Antti E. J. Hyvärinen, and Natasha Sharygina. SolCMC: Solidity compiler's model checker. In Sharon Shoham and Yakir Vizel, editors, *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part I*, volume 13371 of *Lecture Notes in Computer Science*, pages 325–338. Springer, 2022. doi:10.1007/978-3-031-13185-1_16.

8 Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. cvc5: A versatile and industrial-strength smt solver. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 415–442, Cham, 2022. Springer International Publishing.

9 Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org, 2016.

10 Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB): Strings. https://smtlib.cs.uiowa.edu/theories-UnicodeStrings.shtml, 2023.

11 Murphy Berzish. *Z3str4: A Solver for Theories over Strings*. PhD thesis, University of Waterloo, Ontario, Canada, 2021. URL: https://hdl.handle.net/10012/17102.

12 Murphy Berzish, Joel D. Day, Vijay Ganesh, Mitja Kulczynski, Florin Manea, Federico Mora, and Dirk Nowotka. String theories involving regular membership predicates: From practice to theory and back. In Thierry Lecroq and Svetlana Puzynina, editors, *Combinatorics on Words*, pages 50–64, Cham, 2021. Springer International Publishing.

13 Murphy Berzish, Mitja Kulczynski, Federico Mora, Florin Manea, Joel D. Day, Dirk Nowotka, and Vijay Ganesh. An SMT solver for regular expressions and linear arithmetic over string length. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II*, volume 12760 of *Lecture Notes in Computer Science*, pages 289–312. Springer, 2021. doi:10.1007/978-3-030-81688-9_14.

14 František Blahoudek, Yu-Fang Chen, David Chocholatý, Vojtěch Havlena, Lukáš Holík, Ondřej Lengál, and Juraj Síč. Word equations in synergy with regular constraints. In Marsha Chechik, Joost-Pieter Katoen, and Martin Leucker, editors, *Formal Methods*, pages 403–423, Cham, 2023. Springer International Publishing.

15 Véronique Bruyère, Georges Hansel, Christian Michaux, and Roger Villemaire. Logic and p-recognizable sets of integers. *Bulletin of the Belgian Mathematical Society - Simon Stevin*, 1(2):191–238, 1994. doi:10.36045/bbms/1103408547.

16 Taolue Chen, Matthew Hague, Anthony W. Lin, Philipp Rümmer, and Zhilin Wu. Decision procedures for path feasibility of string-manipulating programs with complex operations. *Proc. ACM Program. Lang.*, 3(POPL):49:1–49:30, 2019. doi:10.1145/3290362.

17 Yu-Fang Chen, David Chocholatý, Vojtěch Havlena, Lukáš Holík, Ondřej Lengál, and Juraj Síč. Z3-Noodler: An automata-based string solver. In Bernd Finkbeiner and Laura Kovács, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 24–33, Cham, 2024. Springer Nature Switzerland.

18 Yu-Fang Chen, David Chocholatý, Vojtěch Havlena, Lukáš Holík, Ondřej Lengál, and Juraj Síč. Solving string constraints with lengths by stabilization. *Proc. ACM Program. Lang.*, 7(OOPSLA2), October 2023. doi:10.1145/3622872.

19 Yu-Fang Chen, Vojtech Havlena, Ondrej Lengál, and Andrea Turrini. A symbolic algorithm for the case-split rule in string constraint solving. In Bruno C. d. S. Oliveira, editor, *Programming Languages and Systems - 18th Asian Symposium, APLAS 2020, Fukuoka, Japan, November 30 - December 2, 2020, Proceedings*, volume 12470 of *Lecture Notes in Computer Science*, pages 343–363. Springer, 2020. doi:10.1007/978-3-030-64437-6_18.

20 Yu-Fang Chen, Vojtěch Havlena, Ondřej Lengál, and Andrea Turrini. A symbolic algorithm for the case-split rule in solving word constraints with extensions. *Journal of Systems and Software*, 201:111673, 2023. doi:10.1016/j.jss.2023.111673.

21 Joel D. Day, Thorsten Ehlers, Mitja Kulczynski, Florin Manea, Dirk Nowotka, and Danny Bøgsted Poulsen. On solving word equations using SAT. In Emmanuel Filiot, Raphaël M. Jungers, and Igor Potapov, editors, *Reachability Problems - 13th International Conference, RP 2019, Brussels, Belgium, September 11-13, 2019, Proceedings*, volume 11674 of *Lecture Notes in Computer Science*, pages 93–106. Springer, 2019. doi:10.1007/978-3-030-30806-3_8.

22 Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *TACAS'08*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008. doi:10.1007/978-3-540-78800-3_24.

23 Vijay Ganesh and Murphy Berzish. Undecidability of a theory of strings, linear arithmetic over length, and string-number conversion, 2016. arXiv:1605.09442.

24 Quang Loc Le and Mengda He. A decision procedure for string logic with quadratic equations, regular expressions and length constraints. In Sukyoung Ryu, editor, *Programming Languages and Systems*, pages 350–372, Cham, 2018. Springer International Publishing.

25 Liana Hadarean. String solving at Amazon. https://mosca19.github.io/program/index.html, 2019. Presented at MOSCA'19.

26 G. S. Makanin. The problem of solvability of equations in a free semigroup. *Matematicheskii Sbornik*, 32(2):147–236, 1977. (in Russian).

27 Federico Mora, Murphy Berzish, Mitja Kulczynski, Dirk Nowotka, and Vijay Ganesh. Z3str4: A multi-armed string solver. In Marieke Huisman, Corina S. Pasareanu, and Naijun Zhan, editors, *Formal Methods - 24th International Symposium, FM 2021, Virtual Event, November 20-26, 2021, Proceedings*, volume 13047 of *Lecture Notes in Computer Science*, pages 389–406. Springer, 2021. doi:10.1007/978-3-030-90870-6_21.

28 OWASP. Top 10. https://owasp.org/www-project-top-ten/2017/, 2017.

29 OWASP. Top 10. https://owasp.org/Top10/, 2021.

30 Andrew Reynolds, Andres Nötzli, Clark Barrett, and Cesare Tinelli. A decision procedure for string to code point conversion. In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *Automated Reasoning*, pages 218–237, Cham, 2020. Springer International Publishing.

31 Andrew Reynolds, Andres Notzlit, Clark Barrett, and Cesare Tinelli. Reductions for strings and regular expressions revisited. In *2020 Formal Methods in Computer Aided Design (FMCAD)*, pages 225–235, 2020. doi:10.34727/2020/isbn.978-3-85448-042-6_30.

32 Philipp Rümmer. A constraint sequent calculus for first-order logic with linear integer arithmetic. In *Proceedings, 15th International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, volume 5330 of *LNCS*, pages 274–289. Springer, 2008.

33 Klaus U. Schulz. Makanin's algorithm for word equations-two improvements and a generalization. In Klaus U. Schulz, editor, *Word Equations and Related Topics*, pages 85–150, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg.

34 Z3-Noodler. Automata-based string solver, 2024. URL: https://github.com/VeriFIT/z3-noodler.