The OSTRICH String Solver

Taolue Chen, Riccardo De Masellis, Alejandro Flores-Lamas, Matthew Hague, Zhilei Han, Denghang Hu
Shuanglong Kan, Anthony W. Lin, Oliver Markgraf, Philipp Rümmer, Amanda Stjerna, Zhilin Wu

***Abstract*—This paper gives a high-level overview of the string solver OSTRICH version 1.2, a solver entering SMT-COMP 2022. For more details and theoretical results we refer to the full version of the paper [4] and to the website `https://github.com/uuverifiers/ostrich`.**

I. Overview

OSTRICH is a string solver designed for solving constraints that occur during program analysis. OSTRICH is built on top of the SMT solver Princess [8] and uses the BRICS Automata library [1] to handle regular expressions inside the string formulas. OSTRICH accepts constraints written using the SMT-LIB theory of strings and supports most operators of the theory. The main algorithm implemented in OSTRICH is the background propagation procedure [4], which guarantees completeness of the solver for input formulas in the straight-line fragment of the theory of strings. Some formulas outside of that fragment may still be solved by alternative approaches implemented in OSTRICH, which guarantee soundness (correctness of solutions, and correctness of unsat results), but not completeness.

In addition to standardized SMT-LIB operators, OSTRICH can handle a number of further functions, such as transducers and the string reverse operation. Furthermore, OSTRICH allows users to add their own string functions, as long as they provide an implementation of pre-image computation [4]. OSTRICH can also process regular expressions that include capture groups, lazy quantifiers, and anchors, although this is still at a more experimental stage [3]. For this functionality, OSTRICH understands a number of additional regular expression operators, and it includes a parser for JavaScript regular expression. Those features will not be evaluated, however, at SMT-COMP 2022.

The signature part of the algorithm is the backwards propagation, which is one of the contributions in the main paper on OSTRICH [4]. Additionally, many more optimizations have been implemented, such as a preprocessor and a formula simplifier which are called before and during the main loop of the string solver. OSTRICH was also extended to reason about string formulas by extracting length constraints from those formulas. Finally, more general word equations can be handled by the application of Nielsen's transformation [7], [5].

II. Straight-line Formulas

Straight-line formulas are conjunctions of string constraints that can be represented in the form of straight-line programs S [4]:

$$S \quad ::= \quad x := f(\bar{x}) \mid \textbf{assert}(R(\bar{x})) \mid S; S$$

where f denotes n-ary string functions, for instance concatenation, replace, replace-all, reverse, or any function that can be represented by (one-way or two-way) transducers, and R is a recognisable relation represented by a collection of tuples of finite automata. The statements in a straight-line program have to be sorted topologically, which means that a variable x must never occur prior to an assignment to x.

Example 1. *The following program gives rise to a string formula in the straight-line fragment:*

$$z_1 = x \circ y \ ; \ z_2 = y \circ x \ ; \ \textbf{\textit{assert}}(z_1 \in L \leftrightarrow z_2 \in L)$$

Note that variables z_1, z_2 are only referred to after their definition, whereas x, y are input variables that are never assigned (and therefore might have any value). The assertion $z_1 \in L \leftrightarrow z_2 \in L$ is a recognisable relation. The string formula corresponding to the program is $z_1 = x \circ y \wedge z_2 = y \circ x \wedge (z_1 \in L \leftrightarrow z_2 \in L)$.

We note that straight-line fragments are empirically shown to occupy the bulk of the existing benchmarks. For example, even many length constraints (e.g. $|x| > 20 \vee |y| \leq 30$) turn out to exist in a form that can be converted into regular constraints (a.k.a. monadically decomposable); see [6].

III. Backwards Propagation

Backwards propagation is a decision procedure for straight-line formulas [4]. The main idea of backwards propagation is to systematically compute pre-images of regular expression constraints under the functions occurring in a straight-line program, and this way infer necessary and sufficient constraints on the input variables for the program to succeed. In Example 1, this means that constraints about x, y are derived from the assertion and the two assignments. For a word equation like $z_1 = x \circ y$, constraints about z_1 are propagated to obtain constraints about x, y; a non-deterministic choice has to be made in this process which part of z_1 is assigned to x and which to y, leading to proof branching. Throughout backwards propagation, the consistency of derived constraints for the different variables is checked to assess whether the formula is satisfiable. Backwards propagation also takes into account length information for the string variables, which may lead to an early termination of the branch due to inconsistent word lengths.

IV. Complementary Proof Rules and Optimizations

In OSTRICH, backwards propagation is applied as the last step of the string solver, after all other proof rules have been applied exhaustively to a formula. This is because backwards

propagation is a decision procedure for straight-line formulas, but it cannot be applied to any formula outside of the fragment; backwards propagation also sometimes exhibits high runtime due to the necessary internal splitting. We therefore complement backwards propagation with various other proof rules that have the goal of detecting obvious cases of unsatisfiability, or converting formulas into straightline formulas.

One of the first things to happen in the solver are optimizations in the preproccessor and the formula reducer, which have the goal to simplify string formulas. The applied rewriting rules cover various special cases, as well as many cases of operations that can be reduced to regular language membership or word equations. Examples are functions with trivial input, such as `(str.prefixof x "A")`, which can be turned into a regular constraint about x; or `(str.prefixof x x)`, which can immediately be reduced to `true`. Some reductions are done immediately after parsing the input and are independent of the rest of the formula; other reductions happen during the execution of the string solver ("in-processing"), and are aware of their context. An example of one of the latter transformations is the translation of `(str.prefixof x y)` to word equations, which depends on whether the translated atom occurs positively or negatively.

Prior to calling backwards propagation, OSTRICH also reasons about length constraints induced by the various string operations, and ensures the consistency of such constraints. One example for this are word equations $x = y$, which imply that $|x| = |y|$. This equation about word length can be used to derive a contradiction if it is known that x and y have different lengths. Length information is also used to guide backwards propagation and the use of Nielsen's transformation, described below. Optionally, in addition to length information also induced equations about the number of character occurrences in variables (representing components of the Parikh image) are added.[1]

Nielsen's transformation is applied before the backwards propagation is called, and decomposes word equations into (hopefully) simpler equations. The variant of Nielsen's transformation implemented in OSTRICH applies to string equations $x \circ t = y_1 \circ \cdots \circ y_n$, picks one of the right-hand side variables y_i, and splits the proof into two cases:

$$x = y_1 \circ \cdots \circ y_i' \wedge t = y_i'' \circ \cdots \circ y_n \wedge y_i = y_i' \circ y_i'' \quad (1)$$

$$|x| < |y_1 \circ \cdots \circ y_{i-1}| \vee |x| > |y_1 \circ \cdots \circ y_i| \quad (2)$$

The choice of the variable y_i to split into y_i', y_i'' is done based on the available length information, by first computing a model of all length constraints, and following the arrangement implied by this model [2].

Length constraints are also used to justify the application of more efficient decomposition rules when possible. For example, the word equation $a \circ b = c \circ d$ can directly be decomposed to $a = c \wedge b = d$ if it can be derived that $|a| = |c|$; such decomposition avoids proof branching, and is therefore preferred over Nielsen's transformation.

[1]This can be enabled using option `+parikh`.

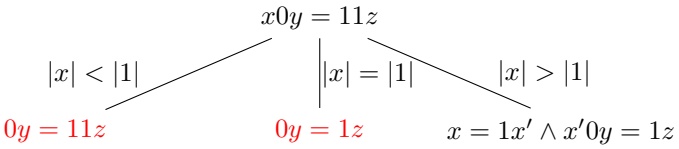

Fig. 1: One step in Nielsen's transformation.

Example 2. *Figure 1 illustrates one step of Nielsen's transformation for the word equation $x0y = 11z$ where x, y, z are string variables and $0, 1$ are string constants. When no length information is available then the transformation guesses the length of the leftmost symbols and decomposes the equation based on the guess. In this example the leftmost branch and the middle branch lead to unsatisfiable word equations and are terminated. The execution of Nielsen's transformation continues then with the right branch and finds a solution $x \mapsto 11, y \mapsto \epsilon, z \mapsto 0$.*

V. OSTRICH AT SMT-COMP 2022

We are submitting the recently released version 1.2 of OSTRICH in the single-query track divisions `QF_S`, `QF_SLIA`, `QF_SNIA`. This version is linked against Princess 2022-07-01 and the BRICS automata library 1.11-8. The submitted version of OSTRICH is configured to use the option `+parikh` to switch on partial Parikh reasoning, see Section IV, otherwise it uses default options.

REFERENCES

[1] Brics automaton. https://www.brics.dk/automaton/index.html, accessed: 2022-06-23

[2] Abdulla, P.A., Atig, M.F., Chen, Y., Holík, L., Rezine, A., Rümmer, P., Stenman, J.: Norn: An SMT solver for string constraints. In: Kroening, D., Pasareanu, C.S. (eds.) Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I. Lecture Notes in Computer Science, vol. 9206, pp. 462–469. Springer (2015). https://doi.org/10.1007/978-3-319-21690-4_29, https://doi.org/10.1007/978-3-319-21690-4_29

[3] Chen, T., Flores-Lamas, A., Hague, M., Han, Z., Hu, D., Kan, S., Lin, A.W., Rümmer, P., Wu, Z.: Solving string constraints with regex-dependent functions through transducers with priorities and variables. Proc. ACM Program. Lang. **6**(POPL), 1–31 (2022). https://doi.org/10.1145/3498707, https://doi.org/10.1145/3498707

[4] Chen, T., Hague, M., Lin, A.W., Rümmer, P., Wu, Z.: Decision procedures for path feasibility of string-manipulating programs with complex operations. Proceedings of the ACM on Programming Languages **3**(POPL), 1–30 (2019)

[5] Diekert, V.: Makanin's Algorithm. In: Lothaire, M. (ed.) Algebraic Combinatorics on Words, Encyclopedia of Mathematics and its Applications, vol. 90, chap. 12, pp. 387–442. Cambridge University Press (2002)

[6] Hague, M., Lin, A.W., Rümmer, P., Wu, Z.: Monadic decomposition in integer linear arithmetic. In: Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part I. pp. 122–140 (2020). https://doi.org/10.1007/978-3-030-51074-9_8, https://doi.org/10.1007/978-3-030-51074-9_8

[7] Lentin, A.: Equations dans les Monoides Libres. Gauthier-Villars, Paris (1972)

[8] Rümmer, P.: A constraint sequent calculus for first-order logic with linear integer arithmetic. In: International Conference on Logic for Programming Artificial Intelligence and Reasoning. pp. 274–289. Springer (2008)