Formal Methods in Computer-Aided Design 2025

S2S: An Eager SMT Solver for Strings

Kevin Lotz
Department of Computer Science
Kiel University
Kiel, Germany
kel@informatik.uni-kiel.de

Mitja Kulczynski
Department of Computer Science
Kiel University
Kiel, Germany
mku@informatik.uni-kiel.de

Dirk Nowotka
Department of Computer Science
Kiel University
Kiel, Germany
dn@informatik.uni-kiel.de

***Abstract*—String constraint solving describes the problem of determining the satisfiability of first-order formulas where variables range over strings. Automated procedures for solving these problems are known as string solvers. Most existing solvers adopt a lazy SMT approach, where a SAT solver handles the Boolean structure of the formula and alternates with a specialized string reasoning engine, following the CDCL(T) paradigm. An alternative strategy, called eager SMT solving, reduces the entire problem to Boolean satisfiability, allowing it to be handled directly by a SAT solver. While successful eager approaches have been proposed, current implementations either lack expressiveness or are not publicly available. Here, we present a new eager string solver based on existing techniques, capable of solving Boolean combinations of word equations, regular constraints, and linear arithmetic over string lengths. An evaluation on the SMT-LIB string benchmarks shows that our approach is competitive on a broad set of problems compared to state-of-the-art solvers, and even outperforms them in many cases. In particular, our solver demonstrates close to best-in-class performance on the SMT-COMP pure string benchmarks.**

I. Introduction

String constraint solving is the task of determining the satisfiability of first-order formulas over string variables, typically involving constructs such as word equations, regular expression membership, and reasoning about string lengths. String constraints are nearly unavoidable in verification tasks and they often arise in security-sensitive domains such as web application analysis and access policy validation [1], [2].

While the satisfiability of word equations and their combination with regular constraints is known to be decidable [3], [4], the known theoretical decision procedures are far from efficient (e.g., [3]–[6]). Moreover, many combinations of string constraints are undecidable (e.g., word equations with string-number conversion predicates [6]) or of unknown decidability status (e.g., word equations with length constraints). Despite these challenges, numerous practical string solvers have been developed. Most solvers employ the *lazy* SMT paradigm, typically by adapting a CDCL(T) approach, which combines a SAT solver for the propositional structure with a dedicated theory solver for string reasoning [7]–[9].

A contrasting approach is the *eager* SMT methodology, in which all constraints are eagerly reduced to propositional logic and handed off to a SAT solver. Two eager solvers for the theory of strings have been proposed: WOORPJE [10] and NFA2SAT [11]. Building on both, we present S2S, a new eager string solver that combines NFA2SAT's incremental reduction approach with WOORPJE's encoding of integer arithmetic. As a result, S2S supports solving Boolean combinations of word equations, regular expression constraints, and linear arithmetic over string lengths and integer variables.

S2S features a two-stage architecture, beginning with a thorough preprocessing phase that applies various simplifications to cut down the search space as much as possible. For the reduction to propositional logic, S2S relies on a bounded approach. It imposes initially small bounds on all variables and gradually increases them when necessary. To make this efficient, the solver adapts the *incremental encodings* and *unsatisfiable core analysis* introduced by NFA2SAT [11]. To handle arithmetic constraints, S2S adopts the approach of WOORPJE, compiling them into propositional logic using multivariate decision diagrams, which compactly represent the sets of bounded integer solutions.

We evaluated S2S on SMT-COMP benchmarks from the *QF_S* and *QF_SLIA* divisions. Results show that S2S is competitive with state-of-the-art solvers OSTRICH [7], Z3 [12], CVC5 [8], and NOODLER [9], and delivers strong performance on both satisfiable and unsatisfiable instances.

II. Solver Architecture

S2S checks the satisfiability of quantifier-free formulas over strings and string lengths, by translating them into propositional logic and solving the encoding with a SAT solver. Currently, S2S supports formulas over the syntax in Figure 2. In addition to Boolean variables, formulas contain variables of sort *string* and sort *integer*; we denote the sets of string and integer variables by $\mathcal{X}_{str}$ and $\mathcal{X}_{int}$, respectively. The atoms of formulas are word equations, regular constraints, linear integer constraints (over string lengths), and prefix, suffix, and containment constraints.

We give an overview of the architecture of S2S, which is sketched in Figure 1. It consists of three major components, the *Context*, a *Preprocessing* pipeline, and the main *Solving Loop*.

A. Context

The context manages variables and their sort, constants, and formulas. Formulas are represented as *abstract syntax trees* (ASTs). All concrete nodes of the structure are stored in a database, and the AST consists only of reference-counted pointers to these nodes. Leaf nodes are constants (strings,

https://doi.org/10.34727/2025/isbn.978-3-85448-084-6_19

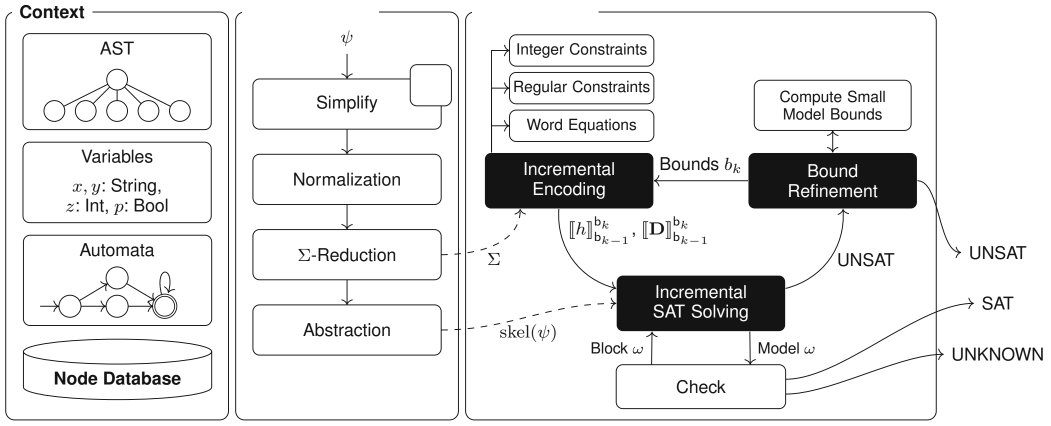

Fig. 1: Architecture of S2S.

$$
\begin{aligned}
F &:= F \vee F \mid F \wedge F \mid \neg F \mid A_{str} \mid A_{int} \mid A_{bool} \\
A_{str} &:= T_{str} \doteq T_{str} \mid T_{str} \mathrel{\dot\in} RE \mid \\
&\quad\ \text{prefix}(T_{str}, T_{str}) \mid \text{suffix}(T_{str}, T_{str}) \mid \text{contains}(T_{str}, T_{str}) \\
A_{int} &:= T_{int} \bowtie T_{int} \ \ (\text{with } \bowtie \in \{<, \leq, =, \geq, >\}) \\
T_{str} &:= T_{str} \cdot T_{str} \mid x \mid w \\
T_{int} &:= T_{int} + T_{int} \mid c \cdot T_{int} \mid c \mid |T_{str}| \mid y
\end{aligned}
$$

Fig. 2: Supported Syntax. Here $\mathcal{X}_{str}$ and $\mathcal{X}_{int}$ are string and integer variables, RE is a regular expression, A_{bool} are Boolean variables, $x \in \mathcal{X}_{str}$, $y \in \mathcal{X}_{int}$, w is constant string, and $c \in \mathbb{Z}$ is a constant integer.

integers, or regular expressions) or variables, all interned in the node database for reuse. Reference-counted pointers enable constant-time cloning and comparison of nodes. To ensure pointer consistency, all nodes are hash-consed upon creation, guaranteeing that no two syntactically identical nodes exist in the database. As a result, the ASTs are internally represented as DAGs.

Nodes are simplified during construction, e.g., by folding constant terms or resolving simple Boolean equivalences. Additionally, the context provides a facade for compiling regular expressions to finite automata. All conversions are cached (again as reference-counted objects), allowing for constant-time retrieval of repeated conversions of the same regular expression.

B. Preprocessing

Before solving a problem, S2S performs several preprocessing steps. First, it optionally simplifies the formula. Then, it prepares the formula for the central solving loop. This involves the conversion into a normal form, thereby removing unsupported constructs, computing a suitable alphabet for the search procedure, and building the Boolean abstraction.

1) Simplification: S2S reduces the search space by running the formula through a pipeline of simplification rules, applied in multiple passes until a fix-point is reached, at which point no further simplifications are possible. Each pass first applies a series of rewrite rules via a post-order traversal of the AST. For each node, the solver searches for an applicable rewrite rule that replace the node with an equivalent but simpler one. Currently, S2S uses 40 rewrite rules, which are applied in a fixed order. These range from Boolean equivalences, to theory-specific rules, such replacing $\alpha\beta \doteq \alpha\gamma$ with $\beta \doteq \gamma$, or identifying that $a\alpha \doteq b\beta$ is unsatisfiable, where α, β, and γ are string terms and a and b are constant characters. All of these rewrites are standard (see, e.g. [10], [13]).

After the rewrite pass, the simplifier tries to infer variable substitutions that preserve satisfiability when applied to the formula. For example, suppose the formula asserts a word equation of the form $xb\alpha \doteq a\beta$ with variable x, constants a, b and arbitrary string terms α, β. The solver detects that necessarily x must start with a and applies the substitution $x \mapsto a \cdot x'$ where x' is a fresh variable. Currently, there are seven rules for inferring substitutions. These rules apply the same reasoning exemplified above to other types of literals, force the value of asserted Boolean variables, or directly set the value of a variable x if it is forced by an asserted equality (e.g., if $x \doteq w$ with $w \in \Sigma^*$, we infer $x \mapsto w$).

The simplifier records all applied substitutions, which are later used for back-substitution when constructing a model for the original formula from a model of the simplified formula.

2) Normalization: The SAT encoding is constructed literal-wise, with a corresponding encoding scheme for every type of literal. To minimize the number of encodings, we transform the formula into a *normal form*, which limits of types of literals occurring in the formula. In this form, the formula is in negation-normal form, and each literal is one of the following

- a word equation $\alpha \doteq \beta$ with string terms α, β, or a dis-equation $x \neq y$ between two string variable $x, y \in \mathcal{X}_{str}$,
- a regular constraint $x \mathrel{\dot\in} R$ or $x \mathrel{\dot\notin} R$ on a single string variable $x \in \mathcal{X}_{str}$,
- one of $\text{prefix}(w, x)$, $\text{suffix}(w, x)$, or $\text{contains}(x, w)$ where $x \in \mathcal{X}_{str}$ and w a constant string, or
- an integer constraint of the form $\sum_{i=1} t_i \cdot c_i \bowtie c$, where each t_i is either an integer variable or the length of a

string variable, $c, c_i \in \mathbb{Z}$ are constant integers, and $\bowtie$ is one of $\{<, \leq, =, \geq, >\}$.

S2S provides an encoding for each type of literal in normal form. Most formulas in Figure 2 can be rewritten into normal form, possibly by introducing fresh variables. For example, a literal of the from $\alpha \dot{\in} R$ is rewritten into a formula in normal form $x \dot{\in} R \wedge x \doteq \alpha$ using a fresh variable x. The only exception are negations of $\mathrm{prefix}(\alpha, \beta)$, $\mathrm{suffix}(\alpha, \beta)$, and $\mathrm{contains}(\alpha, \beta)$ when α is not constant, as they introduce quantification. For example, $\neg\,\mathrm{prefix}(\alpha, \beta)$ is equivalent to $\forall x. \beta \neq \alpha \cdot x$. If α is a constant string w, then we can model this with a regular constraint $\beta \dot{\notin} w \cdot \Sigma^*$, where Σ^* accepts any string. Otherwise, these types of literals are not supported and removed from the formula. The formula in normal form is equivalent (modulo new variables) to the original formula if no literals are dropped.

3) Σ-Reduction: The SMT-LIB theory of strings [14] predefines an alphabet of $3 \cdot 2^{16}$ letters, which is too large for an efficient SAT encoding. Instead, S2S uses the approach of NFA2SAT to construct a subset of this alphabet that preserves satisfiability. The subset consists of all letters occurring in the input problem plus a linear number of extra characters. The exact number of extra characters depends on the formula. If it contains string concatenation, one additional character per inequality [15], otherwise one [11] character per variable.

4) Abstraction: In order to handle Boolean combinations, the solver decomposes the input formula ψ into its propositional structure and theory-specific parts, by replacing every (distinct) first-order atom $a \in \mathrm{atoms}(\psi)$ with a fresh Boolean variable p_a and keeps track of the mapping between a and p_a. As a result, we obtain the *Boolean skeleton* $\mathrm{skel}(\psi)$ of ψ and a set $\mathbf{D}$ of *definitions* of the form $a \rightarrow p_a$ ($\neg a \rightarrow \neg p_a$) if a only occurs in positive (negative) polarity in ψ. We call the formula $\mathrm{skel}(\psi) \wedge \bigwedge_{d \in \mathbf{D}} d$ the *Boolean abstraction* and denote it with $\psi_{\mathcal{A}}$. Based on the results of Plaisted-Greenbaum [16], we have that ψ and $\psi_{\mathcal{A}}$ are equivalent modulo the newly introduced variables in $\psi_{\mathcal{A}}$.

C. Solving Process

The solving process reduces ψ to a propositional formula $[\![\psi]\!]$, which is passed to a SAT solver. To construct $[\![\psi]\!]$, the solver uses a bounded approach: it fixes bounds $\mathsf{b}\colon \mathcal{X}_{str} \cup \mathcal{X}_{int} \rightarrow \mathbb{Z}^2$, where $\mathsf{b}(x) = [l, u]$ denotes the inclusive range of integers $l, \ldots, u$ (with $l \geq 0$ for all $x \in \mathcal{X}_{str}$). The encoding ensures that $[\![\psi]\!]$ is satisfiable precisely if there is a model h of ψ such that $h(x) \in \mathsf{b}(x)$ for all $x \in \mathcal{X}_{int}$ and $|h(x)| \in \mathsf{b}(x)$ for all $x \in \mathcal{X}_{str}$.

Instead of computing precise bounds upfront, S2S starts with small bounds b_1 and gradually increases them. If the encoding $[\![\psi]\!]^{\mathsf{b}_k}$ is unsatisfiable for some bounds b_i, the solver fixes larger bounds b_{k+1} and checks the satisfiability of $[\![\psi]\!]^{\mathsf{b}_{l+1}}$. This results in a series of calls to the SAT solver to determine the satisfiability of the formulas $[\![\psi]\!]^{\mathsf{b}_1}, [\![\psi]\!]^{\mathsf{b}_2}, \ldots,$ each encoding ψ w.r.t. to increasing bounds $\mathsf{b}_1, \mathsf{b}_2, \ldots$.

If for some bounds b_k the SAT solver returns a model ω for $[\![\psi]\!]^{\mathsf{b}_k}$, then S2S decodes ω into a model h for ψ and checks if h also solves the original formula (since the preprocessing steps might have removed literals from the original formula). If yes, S2S reports satisfiability, and optionally the model h. If not, then ω is blocked, and S2S searches for another solution within the same bounds b_k. This is repeated until no further model is found within these bounds, in which case the solver gives up and returns *unknown*.

The formula $[\![\psi]\!]^{\mathsf{b}_k}$ itself is the conjunction

$$\mathrm{skel}(\psi) \wedge [\![h]\!]^{\mathsf{b}_k} \wedge [\![\mathbf{D}]\!]^{\mathsf{b}_k}$$

of the Boolean skeleton $\mathrm{skel}(\psi)$ of ψ, an encoding of the set of possible substitutions $[\![h]\!]^{\mathsf{b}_k}$ over the previously computed alphabet Σ, and encoding of the definitions in $\mathbf{D}$, all bounded by b_k. More precisely, $[\![\mathbf{D}]\!]^{\mathsf{b}_k}$ is the formula

$$\bigwedge_{(p_a \rightarrow a) \in \mathbf{D}} p_a \rightarrow [\![a]\!]^{\mathsf{b}_k} \wedge \bigwedge_{(\neg p_a \rightarrow \neg a) \in \mathbf{D}} \neg p_a \rightarrow [\![\neg a]\!]^{\mathsf{b}_k}$$

where $[\![a]\!]^{\mathsf{b}_k}$ (resp. $[\![\neg a]\!]^{\mathsf{b}_k}$) denote the encoding of a literal in normal form a (resp. $\neg a$).

The encoding depends on the type of the literal. Regular constraints $x \dot{\in} R$ and $x \dot{\notin} R$ are encoded as the reachability problem of nondeterministic finite automata [11], [17]. For word equations $\alpha \doteq \beta$ and their negations, S2S use the encodings introduced in [15]. The encodings prefix, suffix and containment constraints have been are presented in [11]. For linear integer constraint $\sum_{i=1} t_i \cdot c_i \bowtie c$, S2S encodes *multivariate decision-diagrams* similarly to the approach used by WOORPJE [10].

1) Incremental Encoding: To make the iterative search process efficient, we rely on *incremental SAT solving under assumptions* [18] to construct the encodings. All our propositional encodings for literals are built incrementally: the encoding of a literal a at bounds b_k is obtained by only adding clauses to $[\![a]\!]^{\mathsf{b}_{k-1}}$. More precisely, the encoding ensures $[\![a]\!]^{\mathsf{b}_k} \Leftrightarrow [\![a]\!]^{\mathsf{b}_{k-1}} \wedge [\![a]\!]^{\mathsf{b}_k}_{\mathsf{b}_{k-1}}$ for some set of clauses $[\![a]\!]^{\mathsf{b}_k}_{\mathsf{b}_{k-1}}$. In the k^{th} call, it is sufficient to generate only the new clauses $[\![a]\!]^{\mathsf{b}_k}_{\mathsf{b}_{k-1}}$ and add them to the SAT solver, which already retains $[\![a]\!]^{\mathsf{b}_{k-1}}$ from the $k-1^{\text{th}}$ call. Hence, the encoding $[\![\psi]\!]^{\mathsf{b}_k}$ is obtained simply by adding clauses to $[\![\psi]\!]^{\mathsf{b}_{k-1}}$ from the previous iteration.

2) Bound Refinement: If $[\![\psi]\!]^{\mathsf{b}_k}$ is unsatisfiable for some bounds b_k, S2S first checks whether ψ itself is unsatisfiable before updating the bounds. This check relies on the *small-model property* of the theory, which states that every satisfiable formula admits a model of minimal size. If the bounds are large enough to cover such a minimal model, unsatisfiability can be concluded.

To efficiently compute if this is the case, S2S employs the *unsatisfiable core analysis* introduced by NFA2SAT. If an unsatisfiable core C is returned by the SAT solver, S2S constructs a smaller sub-formula ψ^C which is still unsatisfiable but only contains the literals whose encodings are in C. The solver then tries to bound the length of a presumed minimal solution for ψ^C and compares them to b_k. If b_k covers the bounds of the (presumed) minimal solution for all variables,

TABLE I: Results restricted to the fragment S2S supports. The rows Total, Sat, and Unsat correspond the the number on problems solved in that category. The Timeout row shows the number of problems the solver timed out on. Time is given in seconds. The total time does not include timeouts.

	S2S	CVC5	NOODLER	Z3	OSTRICH
			QF_S		
Total	18,782	18,737	**18,889**	18,663	18,739
Sat	11,973	12,003	**12,062**	11,982	11,930
Unsat	**6,809**	6,734	6,827	6,681	**6,809**
Timeout	135	180	**28**	254	177
Time Total	**694.91**	1,034.60	1,506.89	3,762.81	34,579.25
Time SAT	559.12	**550.21**	1,078.76	2,006.85	23,389.17
Time UNSAT	**135.80**	484.39	428.13	1,755.96	11,190.08
			QF_SLIA		
total	47,895	47,303	**48,693**	46,615	45,903
sat	24,398	24,072	**24,740**	23,151	24,098
unsat	23,497	23,231	**23,953**	23,464	21,805
timeout	896	1,490	**80**	2,179	2,828
Time Total	4,196.96	7,067.90	**2,570.70**	9,285.81	118,100.77
Time SAT	3,773.27	5,045.51	**1,511.71**	8,534.13	51,837.43
Time UNSAT	**423.68**	2,022.39	1,058.99	751.68	66,263.34
			QF_S + *QF_SLIA*		
total	66,677	66,040	**67,582**	65,278	64,642
sat	36,371	36,075	**36,802**	35,133	36,028
unsat	30,306	29,965	**30,780**	30,145	28,614
timeout	1,031	1,670	**108**	2,433	3,005
Time Total	4,891.87	8,102.49	**4,077.59**	13,048.62	152,680.02
Time SAT	4,332.39	5,595.71	**2,590.47**	10,540.98	75,226.60
Time UNSAT	**559.48**	2,506.78	1,487.12	2,507.63	77,453.42

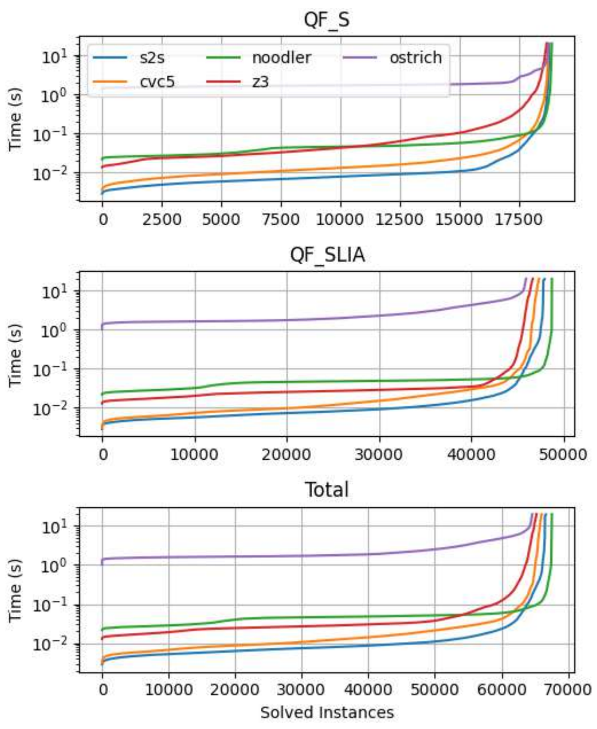

Fig. 3: Cactus plots showing the results on the *QF_S*, *QF_SLIA*, and total benchmarks. Time in seconds (log-scale).

then ψ^C has no minimal solution and, therefore, no solution at all. In that case, S2S reports the unsatisfiability of ψ (since $\psi^C \models \psi$). Otherwise, the solver increases the bounds for all variables in $\mathrm{vars}(\psi^C)$, relaxing all intervals $\mathsf{b}_k(x)$ by a constant offset, without exceeding the computed bounds on the minimal solution.

If ψ contains no word equations and no integer constraints, then S2S always finds bounds on a minimal solution for all variables (see [11]), making it complete for formulas in this fragment. If the formula contains a word equation or integer constraints, S2S relies on the best-effort heuristic described in [15]. This heuristic may fail to establish bounds on the smallest model, in which case unsatisfiability cannot be detected and S2S will increase variable bounds indefinitely without terminating.

III. EVALUATION

The S2S solver is implemented[1] in Rust and uses CADICAL (version 2.1.3) as the backend SAT solver. It accepts problems in the SMT-LIB format. We have evaluated S2S on the 103,335 problems from the SMT-COMP benchmark suite for the theory of strings[2]. These problems are in two categories: *QF_S* (pure string problems, 18,940 in total), and *QF_SLIA* (string problems with linear integer arithmetic, 84,395 in total).

We compare S2S with four state-of-the-art solvers: CVC5 (version 1.2.1), Z3 (version 4.14.0), OSTRICH (version 2024-03-22), and NOODLER (version 4.13.0). To ensure soundness, answers from S2S were verified through majority voting, using the other solvers answers as oracles. Additionally, if S2S returned a model for a satisfiable problem, we double-checked the model's correctness by evaluating it on the original formula.

All experiments were executed on an Ubuntu server equipped with 128 cores and 1.7TiB of memory. We used a timeout of 20 seconds per solver and problem, as used in a variety of prior works on string solvers (see e.g., [10], [19]–[21]). We ran 40 solver instances in parallel. No explicit memory limits were imposed, so all instances shared the available system memory.

S2S solves 18,782 from the *QF_S* (99%) and 47,895 from the *QF_SLIA* (57%) problems, respectively, which corresponds to roughly 65% of all problems. On the remaining problems, the solver returned *unknown*, because they fall out of the supported fragment and no solution can be found when removing the literals that cannot be encoded. The evaluation is limited to problems that S2S can solve. Table I summarizes the results and Figure 3 visualizes them as cactus plots.

S2S solves almost all problems in the *QF_S* category and is the fastest solver overall in that fragment. Only CVC5 solved more satisfiable problems in less time, but it solved fewer problems in total. While NOODLER solved more problems than S2S, S2S was considerably faster, particularly on unsatisfiable instances. Both Z3 and OSTRICH solved fewer problems than S2S and took substantially more time. Notably, S2S was much faster on unsatisfiable problems than any other solver.

[1]Available at https://github.com/s2ssolver/s2s

[2]Benchmarks are available at https://zenodo.org/records/11061097

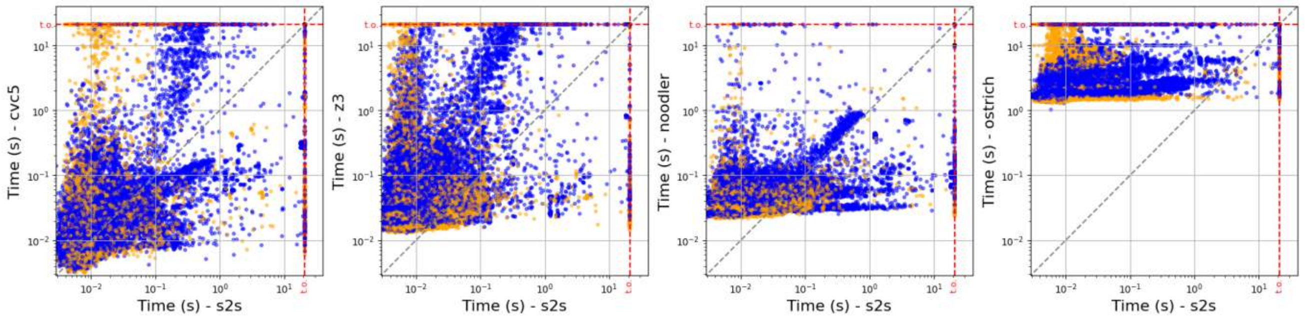

Fig. 4: Scatter plots comparing the runtime of S2S with other solvers on all (*QF_S* + *QF_SLIA*) problems. The axes show the runtime in log-scale. Blue dots are satisfiable and orange dots are unsatisfiable instances. The dashed red lines indicate timeouts.

Restricting to the problems that S2S can solve, we see similar results on the *QF_SLIA* set. In that fragment, S2S is almost twice as fast as CVC5 and solves 492 more problems within the time limit. Among all solvers, NOODLER solved the most problems and was the fastest overall. On satisfiable problems, NOODLER was almost twice as fast as S2S, which ranked second. On unsatisfiable problems, the pattern reversed: S2S was twice as fast as NOODLER, though NOODLER solved considerably more of them. As with the *QF_S* problems, Z3 and OSTRICH solved fewer problems and took more time than the other solvers.

The scatter plots in Figure 4 give a more detailed comparison between the solvers. They compare the runtimes of S2S (x-axis) to the runtimes of CVC5, Z3, NOODLER, and OSTRICH, respectively (y-axis). Every dot represents a single problem, with the axes indicating the runtime of each solver for that problem. Blue dots correspond to satisfiable problems, while yellow dots indicate unsatisfiable ones. A dot is above the diagonal precisely if S2S solved the problem faster. We can see that, independently of the solver we compare to, most orange dots are above the diagonal. This aligns with the findings that S2S is generally fast on unsatisfiable instances.

We can see that almost all solvers need a constant amount of time before solving any problem, visible as a gap between the axis and the lowest data point. We refer to this as the *startup time*. The solvers Z3, NOODLER (which is based on Z3), and OSTRICHhave relatively high startup time, compared to S2S, which show little to no startup time. CVC5 has only a slightly higher startup than S2Sand for a small subset that CVC5 solves fast, it has little to no startup time at all. We reckon that these might be instances that can be solved merely by simplification, without starting the search procedure.

IV. CONCLUSION

We presented S2S, an eager string solver that reduces string constraints to propositional logic and solves them via incremental SAT solving. The solver combines the ideas of NFA2SAT and WOORPJE. Despite supporting only a fragment of the full SMT-LIB theory, S2S achieves strong performance on a large fraction of the SMT-COMP benchmarks. Unlike lazy solvers, S2S avoids dedicated theory engines and offers an alternative for the supported fragment.

We plan to extend expressiveness by support to additional types of literals, such as transduction. This requires finding efficient SAT encodings that integrate with the incremental solving approach. Another possible research direction is to explore more sophisticated CEGAR approaches. Using appropriate over-approximations of the input formula could lead to more efficient SAT encodings and significantly improve the solver's performance. Finally, we believe that bound refinement procedure can be improved through careful in-processing, likely resulting in further performance improvements.

REFERENCES

[1] T. Bultan, F. Yu, M. Alkhalaf, and A. Aydin, *String Analysis for Software Verification and Security*. Springer International Publishing, 2017. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-68670-7

[2] J. Backes, P. Bolignano, B. Cook, C. Dodge, A. Gacek, K. Luckow, N. Rungta, O. Tkachuk, and C. Varming, "Semantic-based automated reasoning for AWS access policies using SMT," in *2018 Formal Methods in Computer Aided Design (FMCAD)*, 2018, pp. 1–9.

[3] G. S. Makanin, "The problem of solvability of equations in a free semigroup," *Mathematics of the USSR-Sbornik*, vol. 32, no. 2, pp. 129–198, Feb. 1977. [Online]. Available: http://dx.doi.org/10.1070/SM1977v032n02ABEH002376

[4] K. U. Schulz, "Makanin's algorithm for word equations-two improvements and a generalization," in *Word Equations and Related Topics*, K. U. Schulz, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 85–150.

[5] M. Berzish, J. D. Day, V. Ganesh, M. Kulczynski, F. Manea, F. Mora, and D. Nowotka, "String theories involving regular membership predicates: From practice to theory and back," in *Combinatorics on Words*, T. Lecroq and S. Puzynina, Eds. Cham: Springer International Publishing, 2021, pp. 50–64.

[6] J. D. Day, V. Ganesh, P. He, F. Manea, and D. Nowotka, "The satisfiability of word equations: Decidable and undecidable theories," in *Reachability Problems*, I. Potapov and P.-A. Reynier, Eds. Cham: Springer International Publishing, 2018, pp. 15–29.

[7] T. Chen, M. Hague, A. W. Lin, P. Rümmer, and Z. Wu, "Decision procedures for path feasibility of string-manipulating programs with complex operations," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, Jan. 2019. [Online]. Available: https://doi.org/10.1145/3290362

[8] H. Barbosa, C. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, and Y. Zohar, "cvc5: A versatile and industrial-strength SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, D. Fisman and G. Rosu, Eds. Cham: Springer International Publishing, 2022, pp. 415–442.

[9] Y.-F. Chen, D. Chocholatý, V. Havlena, L. Holík, O. Lengál, and J. Síč, "Z3-noodler: An automata-based string solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, B. Finkbeiner and L. Kovács, Eds. Cham: Springer Nature Switzerland, 2024, pp. 24–33.

[10] J. D. Day, T. Ehlers, M. Kulczynski, F. Manea, D. Nowotka, and D. B. Poulsen, "On solving word equations using SAT," in *Reachability Problems - 13th International Conference, RP 2019, Brussels, Belgium, September 11-13, 2019, Proceedings*, ser. Lecture Notes in Computer Science, E. Filiot, R. M. Jungers, and I. Potapov, Eds., vol. 11674. Springer, 2019, pp. 93–106. [Online]. Available: https://doi.org/10.1007/978-3-030-30806-3_8

[11] K. Lotz, A. Goel, B. Dutertre, B. Kiesl-Reiter, S. Kong, R. Majumdar, and D. Nowotka, "Solving string constraints using SAT," in *Computer Aided Verification*, C. Enea and A. Lal, Eds. Cham: Springer Nature Switzerland, 2023, pp. 187–208.

[12] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340.

[13] A. Reynolds, M. Woo, C. Barrett, D. Brumley, T. Liang, and C. Tinelli, "Scaling up DPLL (T) string solvers using context-dependent simplification," in *International Conference on Computer Aided Verification*. Springer, 2017, pp. 453–474.

[14] C. Tinelli, C. Barrett, and P. Fontaine, "SMT-LIB theory of unicode strings," 2020. [Online]. Available: https://smt-lib.org/theories-UnicodeStrings.shtml

[15] K. Lotz, A. Goel, B. Dutertre, B. Kiesl-Reiter, S. Kong, and D. Nowotka, "Solving string constraints with concatenation using SAT," in *Formal Methods in Computer-Aided Design*, N. Narodytska and P. Rümmer, Eds. TU Wien Academic Press, 2024.

[16] D. A. Plaisted and S. Greenbaum, "A structure-preserving clause form translation," *Journal of Symbolic Computation*, vol. 2, no. 3, pp. 293–304, 1986. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0747717186800281

[17] M. Kulczynski, K. Lotz, D. Nowotka, and D. B. Poulsen, "Solving string theories involving regular membership predicates using SAT," in *Model Checking Software*, O. Legunsen and G. Rosu, Eds. Cham: Springer International Publishing, 2022, pp. 134–151.

[18] N. Eén and N. Sörensson, "Temporal induction by incremental SAT solving," *Electronic Notes in Theoretical Computer Science*, vol. 89, no. 4, pp. 543–560, 2003, bMC'2003, First International Workshop on Bounded Model Checking. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1571066105825423

[19] M. Berzish, V. Ganesh, and Y. Zheng, "Z3str3: A string solver with theory-aware heuristics," in *2017 Formal Methods in Computer Aided Design (FMCAD)*, 2017, pp. 55–59.

[20] F. Mora, M. Berzish, M. Kulczynski, D. Nowotka, and V. Ganesh, "Z3str4: A multi-armed string solver," in *Formal Methods: 24th International Symposium, FM 2021, Virtual Event, November 20-26, 2021, Proceedings*. Berlin, Heidelberg: Springer-Verlag, 2021, pp. 389–406. [Online]. Available: https://doi.org/10.1007/978-3-030-90870-6_21

[21] M. Kulczynski, F. Manea, D. Nowotka, and D. B. Poulsen, "The power of string solving: Simplicity of comparison," in *Proceedings of the IEEE/ACM 1st International Conference on Automation of Software Test*, ser. AST '20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 85–88. [Online]. Available: https://doi.org/10.1145/3387903.3389317