BASC: Benchmark Analysis for String Constraints

Joel D. Day[1], Adrian Kröger[2], Mitja Kulczynski[3], Florin Manea[2], Dirk Nowotka[3] and Danny Bøgsted Poulsen[4]

[1] Department of Computer Science, Loughborough University, Loughborough, UK
[2] Department of Computer Science, University of Göttingen, Göttingen, Germany
[3] Department of Computer Science, Kiel University, Kiel, Germany
[4] Department of Computer Science, Aalborg University, Aalborg, Denmark

Abstract. String constraint solving, and the underlying theory of word equations, are highly interesting research topics both for practitioners and theoreticians working in the wide area of satisfiability modulo theories. As string constraint solving algorithms, a.k.a. string solvers, gained a more prominent role in the formal analysis of string-heavy programs, especially in connection to symbolic code execution and security protocol verification, we can witness an ever-growing number of benchmarks collecting string solving instances from real-world applications as well as an ever-growing need for more efficient and reliable solvers, especially for the aforementioned real-world instances. Thus, it seems that the string solving area (and the developers, theoreticians, and end-users active in it) could greatly benefit from a better understanding and processing of the existing string solving benchmarks. In this context, we propose BASC: a Benchmark Analysis tool for String Constraints. BASC is implemented in PYTHON 3, and offers a collection of analysis and information extraction tools for a comprehensive data base of string benchmarks (presented in SMT-LIB format), based on an SQL-centred language called QLANG.

1 Introduction

The theory of string solving is a research area in which one is interested in the mathematical and algorithmic properties of systems of constraints involving (but not restricted to) string variables and string constants. As such, string solving is part of the general constraint satisfiability topic, where one is interested in the satisfiability of formulae modulo logical theories over strings. Recent motivations for theoretical and practical investigations in this area come from the verification of security protocols (e.g., detecting security flaws exploited in *injection attacks* or *cross site scripting attacks*) or the symbolic execution of string-heavy languages. Excellent overviews of main definitions and results, as well as of the many recent developments related to the theory and practice of string solving, are [1, 8].

Relevant to our work, on the practical side, a series of dedicated string constraint solvers were developed (see [1]), but also well-established general-purpose SMT solvers (such as CVC4 [2] and Z3 [7, 12]) started offering integrated string solving components. The efforts dedicated to improving the performance of many of these solvers are still ongoing.

Thus, having a reliable and curated collection of benchmarks containing string constraints seems to be of foremost importance for the development and evaluation of string solvers. The main benchmarks used in the evaluation of string solvers are presented in detail in [11][1] and they were extracted both from real-world and artificial scenarios. For examples of systems of string constraints, as found in the benchmarks, see, e.g., [3, 14][1]. Moreover, there exists now a unified string-logic standard as part of SMT-LIB, and the tool ZaligVinder [11] brings together a set of relevant benchmarks and introduces a uniform benchmarking framework. Nevertheless, there are still some challenges related to string-solving benchmarks. Firstly, they are mostly uncategorized with respect to the type of string constraints they contain, and solvers addressing specific types of constraints have to first pre-process the existing benchmarks and extract the relevant constraints [4, 6]. Secondly, the performance of solvers on the benchmarks was sometimes hard to observe and compare: the sat or unsat answers provided by some existing tools were sometimes wrong on a relatively large set of inputs [10], and the size of benchmarks means it is challenging to get an overview on where one solver outperforms others and perhaps even more importantly, why.

Research Tasks. In this context, we can formulate a series of research tasks addressing the main issues related to the string solving benchmarks.

1. Identify, store, and organize a comprehensive collection of benchmarks for string solving as a database, allowing querying, export, and information extraction from the benchmarks, as well as an interface for running supported string solvers on specific benchmarks, extracted w.r.t. certain requirements from the entire database, and evaluating their performance.
2. Offer functionalities allowing the extension of the database with new benchmarks, as well as the integration of new string solvers.

A tool answering these questions would be the first database tool in the area of string solving which allows the extraction of information from string solving benchmarks and fair and uniform comparison of string solvers on a selected set of benchmarks displaying certain particularities. Such a tool could also open the way towards deeper research tasks related to the evaluation of the solvers' performance, such as analysing the impact that the preprocessing part executed by a solver has on the performance, or integrating external tools in the database, allowing the generation of new instances based on existing benchmarks. Also, such a tool would fit in the direction of creating large collections of benchmarks containing SMT or SAT instances [14, 9].

Our contribution. We propose BASC: a Benchmark Analysis tool for String Constraints. It is accessible at: `https://b4sc.github.io`.

BASC is implemented in Python 3, and offers a collection of analysis and information extraction tools for the most comprehensive database of string benchmarks (collected from the literature and presented in SMT-LIB 2.5 format), based on an SQL-centred language called qlang. Besides basic database man-

[1] see Appendix A.1 for more details

agement, benchmark querying, and analysis capabilities, BASC also offers an interface to running and testing string solvers on the benchmarks. The results of such runs can then be collected, stored, further analyzed, and correlated to other properties of the respective benchmarks (computed using BASC database queries). As such, BASC offers solutions to our two research tasks. BASC also offers to users a simple method for implementing and running their own analysis on the benchmarks, as well as the possibility of collecting and integrating the results of this analysis into the database. Implementation details of BASC are discussed in the rest of the paper and in the documentation of the tool.
User Base. BASC is, in our opinion, beneficial to the following three communities of potential users. Firstly, the community of string solver developers, for which it eases the performance-analysis of specific solvers, on specific benchmarks, and, thus, helps identify the strengths and weaknesses of each string solver. Secondly, the theory community: BASC facilitates the further understanding of theoretical properties of specific classes of word equations, relevant in practice. Finally, the end-users: entities who have (or develop) use cases with string constraints, of relevance to their activities, and want to understand better the nature of these benchmarks w.r.t. standard structural measures for string constraints, or solve their instances as correctly and efficiently as possible; for them, producing their own analysis tools or solvers could be too expensive so they could integrate their cases in BASC, and use the offered methods to analyze it.[2]

2 Architecture of BASC

BASC provides a series of mechanisms easing the access to a comprehensive set of benchmarks, based on an SQL-inspired query language called QLANG. The queries expressible in QLANG allow accessing and analysing rehashed and preprocessed benchmarks. To implement the main structure of our database of string constraints-benchmarks, we proceeded as follows. The central information is stored in an SQLITE database which consists of five different tables, namely `benchmark`, `tracks`, `instances`, `results`, and `validated_results`. Each benchmark set contains multiple tracks, e.g., KALUZA contains different tracks, grouping instances w.r.t. their size and satisfiability, which is reflected in our database structure via tables `benchmark` and `tracks`. A track itself contains multiple linked instances, stored within the `instances` table. Thus, the `instances` table stores for each record a file path and additional information, such as a unique name. Since we also provide an interface for running different solvers on the available benchmarks, we allow storing the results of these runs in the database in the `results` table. These results are cross validated w.r.t. the existing solvers (as explained in the online documentation of BASC) and the conclusions are stored in table `validated_results`. We provide an easy interface allowing to add additional solvers. Currently, BASC implements it for CVC4, Z3STR3, and Z3SEQ, but can be extended to include string solvers capable of reading/processing SMT-LIB files. To achieve this, we reused the engine of ZALIGVINDER.

[2] In the Appendix A.2, we overview several concrete application scenarios.

$$\begin{array}{lcl}
S & ::= & \texttt{Select}\ f_s\ \texttt{From}\ d\ \texttt{Where}\ c \ \mid\ \texttt{Extract}\ f_e\ \texttt{From}\ d\ \texttt{Where}\ c\ \texttt{Apply}\ \textsc{Function} \\
f_s & ::= & \texttt{Name} \ \mid\ \texttt{Hash} \ \mid\ \texttt{Content} \\
f_e & ::= & \texttt{SMTLib} \ \mid\ \texttt{SMTPlot} \ \mid\ \ldots \\
d & ::= & * \ \mid\ \textsc{Set} \ \mid\ \textsc{Set:Track} \ \mid\ d,\ d \\
c & ::= & \textsc{Predicate} \ \mid\ (c\ \texttt{And}\ c) \ \mid\ (c\ \texttt{Or}\ c) \ \mid\ (\texttt{Not}\ c) \ \mid\ \texttt{True} \ \mid\ \texttt{False}
\end{array}$$

Fig. 1. Syntax of QLANG.

To allow gathering new insights about the benchmarks, BASC offers an interface permitting the definition of custom benchmark-analysis predicates, which can directly interact with the SMT-LIB instances and the precalculated information regarding them. To this end, for each instance contained in the database, we additionally store an Abstract Syntax Tree (AST) within the file-system, and have the possibility to augment each node of the tree with additional information. These ASTs are the fundamental data-structures used in our approach.

The language QLANG: Let us now go a bit more into details regarding the query language QLANG. As main functionality, this language allows the selection of instances (i.e., printing file names matching a query or exporting them after potentially applying a modification). The syntax of QLANG is given in Figure 1.

The semantic of a **Select** query is based on the data set d. We either choose all benchmarks ($*$) or we are more precise in picking a particular benchmark set respectively a corresponding track. The selection is based on a Boolean expression c, constructed from basic PREDICATEs. Currently, BASC implements several default predicates.[3] For instance, `hasWEQ` selects the benchmarks containing at least one word equation, or `isSAT(`*solver*`)` returns instances being declared as sat by the particular *solver*. Worth noting, our interface allows also defining custom predicates. As far as the implementation is concerned, when evaluating a predicate on an instance, this predicate is applied bottom-up to the corresponding AST and its value is evaluated in the root. Therefore, in the case of custom predicates, the user has to specify two functions, namely an `apply`- and a `merge`-function. The `apply`-function specifies how the actual computation of the predicate is done for the information corresponding to a single node, while the `merge`-function processes the data computed by the children of the argument node. The actual predicate uses the computed data for an AST and returns either `true` or `false`, thus allowing the selection of particular instances based on this return value. We use f_s to choose a suitable output which can be the instance name, the file's hash value, or simply the SMT-LIB instance.

The **Extract** query allows exporting instances for which a Boolean expression (again involving predicates) evaluates to `true`, just as described above. Additionally, while executing an **Extract** query, we can directly perform modifications to the extracted instances using a `Function`. These functions are applied (similarly to the case of predicates) node-wise, bottom-up, on the ASTs corresponding to the processed instances. Therefore, for such queries, the user specifies for the nodes an `apply`-function, which performs the modification of a node. This technique allows, for example, applying simplification rules to specific nodes. Moreover, this interface allows the application of external procedures on the whole AST, e.g. permitting us to apply a fuzzer like STRINGFUZZ [5] to generate new instances having a similar structure to the extracted ones. Finally, the argument f_e of an **Extract** query specifies the output format for the matching (and potentially modified) instances, e.g. in SMT-LIB format or a plot visualizing the tree-like

$$
\begin{array}{lll}
\mathit{Expr} & ::= & (\mathit{Id},\ \mathit{Kind},\ \mathit{Decl},\ \mathit{Sort},\ \mathit{Params},\ \mathit{Children},\ \textsc{IntelDictionary}) \\
\mathit{Id} & ::= & x \quad \text{for } x \in \mathbb{N} \qquad\qquad \mathit{Kind} ::= \texttt{Variable} \mid \texttt{Other} \\
\mathit{Decl} & ::= & \texttt{and} \mid \texttt{or} \mid \texttt{not} \mid \ldots \mid \texttt{=} \mid \texttt{<=} \mid \ldots \mid \texttt{substr} \\
\mathit{Sort} & ::= & \texttt{String} \mid \texttt{Bool} \mid \texttt{RE} \mid \texttt{Integer} \\
\mathit{Params} & ::= & [p] \mid [] \qquad\qquad p ::= x \mid p,\ p \quad \text{for } x \in \mathbb{Z} \\
\mathit{Children} & ::= & [c] \mid [] \qquad\qquad c ::= \mathit{Expr} \mid c,\ c
\end{array}
$$

Fig. 2. Internal representation of string constraints.

structure. As a simple example, to count instances which exhibit certain properties, one could use the predefined operation `Count`.[3] Notably, a function might also translate ASTs into different (not necessarily tree-like) structures, providing, in a sense, an interpretation of these trees suitable to the desired application.

As an example for this section, to obtain a list of all benchmarks containing word equations and determined satisfiable by the string solver CVC4, we can execute the query **Select** `Name` **From** `*` **Where** `(hasWEQ` **and** `isSAT(CVC4))`.[3]

The AST structure: In the center of our implementation is the AST data structure, which is directly derived from the SMT-LIB instances. A string constraint defined in SMT-LIB starts with a block declaring all variables (note that the string constraints are not quantified in the respective standard). All constraints that have to be satisfied at the same time follow these declarations. Based on this structure, our AST is a parse tree derived according to the formal grammar from Figure 2. As it can be seen there, BASC currently supports common string-constraints: word equations, arithmetic-over-lengths, regular-language-membership, Boolean constraints. We use Z3 as input-parser for SMT-LIB, so BASC parses all constraints Z3 handles. Our internal representation uses a generic expression to represent constraints/types not covered explicitly yet. Thus, BASC can be easily extended to address other constraint-types.

Each expression *Expr* has a unique *Id*, a named operator which can essentially be any operator available in the SMT-LIB for string constraints, a *Kind* declaring whether the given expression is a single variable or not, and a *Sort*. Additionally, an expression might have additional parameters; for instance, `re.loop` which corresponds to a bounded Kleene star operation w.r.t. the parameters. Furthermore, an expression can have multiple children which are again expressions. Finally, each expression stores a unique IntelDictionary containing all the information computed using the previously introduced predicates and stored in the database. This structure allows accessing individual nodes quickly. To avoid recalculating the ASTs over and over again, whenever needed, we use Pickle [13] to store the tree within the file system. As such, an AST corresponding to a particular instance (file) is available and can be enriched at any time.

One key aspect of BASC's architecture is the usage of the ASTs in the definition of predicates, functions, and extractors. While defining meaningful and efficient predicates/functions is an algorithmic problem, which needs to be addressed individually for each predicate/function, our architecture offers both a fundamental data structure, easily and naturally adaptable to specific scenarios, as well as an accessible interface for defining and implementing those functions.

Multiple examples for using our tool and further implementation details are given in its documentation.[4]

[3] See Appendices A.4 and A.5 for more examples.

[4] To ease the reviewing process, some examples are given in the Appendix.

3 Conclusion and Future Work:

We introduced a benchmark analysis framework called BASC to analyse string constraints and string solvers. Our toolbox provides a query language allowing the exploration of a custom benchmark set. BASC provides several useful functions, predicates, and extractors for straight use, within custom queries, and we are continuously working towards enriching the current toolbox with more such operations. Other directions of development are to also offer built-in coverage for more general theories (e.g., including bitvectors), which are currently treated as generic, as well as to offer good mechanisms for debugging, logging, and diagnostics, especially in the context of user-defined functions.

References

1. Amadini, R.: A survey on string constraint solving. ACM Computing Surveys (CSUR) **55**(1), 1–38 (2021)
2. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: CAV 2011. pp. 171–177. Springer (2011)
3. Barrett, C., Stump, A., Tinelli, C., et al.: The SMT-lib standard: Version 2.5. Available at: **https://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.5-r2015-06-28.pdf** (2015)
4. Berzish, M., Kulczynski, M., Mora, F., Manea, F., Day, J.D., Nowotka, D., Ganesh, V.: An SMT Solver for Regular Expressions and Linear Arithmetic over String Length. In: CAV 2021. pp. 289–312. Springer (2021)
5. Blotsky, D., Mora, F., Berzish, M., Zheng, Y., Kabir, I., Ganesh, V.: Stringfuzz: A fuzzer for string solvers. In: CAV 2018. pp. 45–51. Springer (2018)
6. 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. CoRR **abs/2111.04298** (2021, to appear in POPL 2022)
7. De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: TACAS 2008. pp. 337–340. Springer (2008)
8. Hague, M.: Strings at MOSCA. ACM SIGLOG News **6**(4), 4–22 (2019)
9. Heule, M., Jarvisalo, M., Suda, M., Iser, M., Balyo, T., Froleyks, N.: SAT Competition '21 benchmarks. `https://satcompetition.github.io/2021/downloads.html`, accessed: 2022-01-17
10. Kulczynski, M.: Light On String Solving: Approaches to Efficiently and Correctly Solving String Constraints. Ph.D. thesis (2022)
11. Kulczynski, M., Manea, F., Nowotka, D., Poulsen, D.B.: ZaligVinder: A generic test framework for string solvers. J. Software: Evolution and Process p. e2400 (2021)
12. Mora, F., Berzish, M., Kulczynski, M., Nowotka, D., Ganesh, V.: Z3str4: A Multi-armed String Solver. In: FM 2021. pp. 389–406. Springer (2021)
13. Python Software Foundation: pickle — python object serialization. `https://docs.python.org/3/library/pickle.html`, accessed: 2022-01-21
14. The SMT-LIB Initiative: The SMT Library. `https://smtlib.cs.uiowa.edu/benchmarks.shtml`, accessed: 2022-01-17

A Appendix

For clarity, we restate here the link to BASC:
`https://b4sc.github.io/`
The bibliographic references in the Appendix refer to the list at its end (and they are given in a different style to avoid confusions).

A.1 String Solvers, Benchmarks and an example of SMT-Lib instance

To get a better image of the developments occurring in string solving, we recall first a series of important dedicated string constraint solvers: Hampi [KGG+09], Norn [AAC+15], Stranger [YAB10], ABC [ABB15], Woorpje [DEK+19], [DKM+20], OSTRICH [CHL+19], CertiStr [KLRS22].

Also, well-established general-purpose SMT solvers, like CVC4 [BCD+11] and Z3 [DMB08,MBK+21], started offering integrated string solving components.

The main benchmarks used in the evaluation of string solvers are presented in detail in [KMNP21]. These benchmarks were extracted both from real-world and from artificial scenarios. Some benchmarks based on real-world scenarios are related to, e.g., symbolic execution of string heavy programs (Kaluza, PyEx, LeetCode), software verification (Norn), sanitization (PISA), or to detection of software vulnerabilities caused by improper string manipulation (AppScan, JOACO, Stranger). Some artificially produced benchmarks are based either on theoretical insights (Sloth, Woorpje, Light Trau) or on fuzzing algorithms (BanditFuzz, StringFuzz).

```
(set-logic QF_SLIA)
(declare-fun v1 () String)
(declare-fun v2 () String)
(declare-fun v3 () Int)
(declare-fun ret () String)
(assert (= v2 "<") )
(assert (ite (str.contains v1 v2) (and (= v3 (str.indexof v1 v2 0)) (= ret (str
    ↪ .substr v1 0 v3)))(= ret v1)))
(assert (or (str.contains ret "<")  (str.contains ret ">")))
(check-sat)
```

Listing 1.1. Instance taken from the PISA set

In general, all these benchmarks contain systems of string constraints. For instance, in Listing 1.1 we depict an instance from the PISA [ZZG13] set. It models a conditional choice of a return variable `ret` making assumptions on other variables having the data type strings.

For more examples of systems of string constraints, as found in the benchmarks, see [BST+10,BST+15,ZZG13,The].

A.2 Potential Applications of BASC

We begin with clear examples from the literature, where this tool could have been used. Then we give three examples of problems which could have solutions based on BASC.

There is demonstrable need for analyses of benchmarks' properties. In the following cases, hand-crafted benchmarks and ad-hoc analyses were created and used:

- An example for ad-hoc analysis is [CCH$^+$18] which motivates the Straight-line fragment of string constraints by noting that the Kaluza benchmark set falls within it. As the Kaluza benchmarks become older and new sets emerge, it would be beneficial to have updated analyses, so that assumptions about practical instances do not become out-of-date. BASC will make these much quicker and easier.
- An example for hand-crafted benchmarks is [LH18]. This paper could have benefited from our tool. The authors argue that a new, hand-crafted benchmark, whose instances fulfil some specific properties, is required in order to be able to compare their decision procedure with existing solvers. In such cases, it would be faster and more transparent to use BASC to extract from existing benchmarks the instances exhibiting the addressed properties, and compare on those.
- A similar argument can be made related to [CFH$^+$22], where the authors say "There are no standard string benchmarks involving RegExes[...]". Using BASC such benchmarks could be extracted from existing instances, and relevant new benchmarks could be added to the database.
- Nevertheless, in [BKM$^+$21], a set of over 100000 benchmarks was analyzed ad-hoc, to extract string constraints containing only regexes and linear arithmetic and detect their structural-complexity, with the aim of producing an efficient solver. This required a complex and tedious approach. Using our database and BASC-features this can be done much easier.

Therefore, it seems that analyses like those from the aforementoned papers [CCH$^+$18,LH18,BDG$^+$21,BKM$^+$21,CFH$^+$22,CHL$^+$19] require significant effort, replicated every time a new analysis is needed. We expect BASC to be used routinely by developers of string-solvers (see [Ama21,Hag19] for pointers to such groups) to provide faster and transparent evaluation, up-to-date context, and applicability-evidence for their algorithms.

Similarly, many theoreticians use string-solving as motivation for their works. BASC can provide real evidence supporting this and also inform future directions of study. [Hag19] contains examples of research-groups likely to use BASC.

We now move to some more examples of problems which can be addressed with BASC. The first two examples and concrete implementations for them are also discussed on the webpage of BASC.

Example 1.
Problem: Given a syntactically restricted subset of string constraints, determine instances belonging to this subset, and their distribution w.r.t. benchmarks.
Motivation: many theory papers provide insightful results (algorithms, complexity bounds, information about solution-sets) into subclasses of string constraints [Hag19]. Such subclasses include those defined directly (e.g., constraints in solved-form, acyclic or straight-line constraints) as well as indirectly (e.g.,

quadratic/regular word equations). Knowing how applicable such results/insights are in practice, and thus whether it makes sense to incorporate them in the design and implementation of string solvers, requires first knowing how many string constraints belong to those subclasses. By solving this, BASC has value for researchers approaching subcases of word equations/automata/arithmetic, who could use our tool to see if that subcase is relevant to string-constraint solving in practice, and if so, provide evidence of this as motivation for their work. If the defined subcase is not prominent, they could use it to guide changes to the definition with the goal of making it more applicable while preserving theoretical properties. It is also of value to researchers developing string solvers who will benefit from knowing (A) which theoretical insights are most likely to be effective over a broad range of use-cases and (B) which cases/properties to target with their own optimizations and innovations.
Technical challenges: firstly, syntactic restrictions continually arise from a variety of (theory-)sources and will not necessarily be formulated directly in the usual nomenclature of string constraints and SMT-solvers. Consequently, simply deciding whether a string constraint belongs to a subclass of interest can range from trivial to requiring substantial processing. For example, it is not immediately clear given a single instance, whether e.g. the systems of word equations arising when solving it are all quadratic. Secondly, the set of benchmarks is also regularly being expanded and updated (and some might also depreciate). Thirdly, many modern string solvers rewrite string constraints in a preprocessing stage. Obtaining realistic data therefore requires taking into account the effects of rewriting processes in relation to syntactic subsets.
Novelty: currently, there are no tools capable of properly addressing this problem and these challenges. Without BASC, understanding e.g. how many string constraints belong to various relevant fragments is something which would have required significant effort for just a single case. Note that these analyses have not been yet carried out even for major subclasses of string constraints.

Example 2.
Problem: For a given string-solver, understand the properties of instances on which it performs particularly well, and on which it performs badly.
Motivation: such insights are valuable for designing new and improving/optimizing existing string solvers. It is also valuable for constructing *portfolio solvers.*
Technical challenges: Clearly, some challenges stem from the same issues discussed in Example 1 above. Moreover, once we computed a set of instances on which a solver performs well and a set of instances on which that solver performs badly, we need reliable analysis tools to find properties which separate the two sets.
Novelty: Tools exist already which assist the comparison of string-solvers in terms of directly evaluating how they perform over a set or sets of benchmarks (e.g., [KMNP21]). This is sufficient for producing evidence of their effectiveness, w.r.t. state-of-the-art. However, without BASC, it is difficult even to understand the character of particular benchmarks beyond very superficial observations. Thus, existing tools say little about why or when a given solver performs well. Our

tool is the first to facilitate analyses of the form "Solver X performs best on string constraints containing complex word equations" where "complex" can be formally defined by some well-motivated criteria.

Our third example is more exploratory, and it shows the potential of future research based on BASC.

Example 3.
Problem: Develop a learning-algorithm which detects the best solver, from a given set, for each instance.
Motivation: such a method would be very useful in the construction of *portfolio solvers*. Also, according to the learning-algorithms used, we could gain a lot of insight into the strengths of the solvers. For instance, we could obtain decision-trees guiding the selection of solvers on a certain data, according to some numerical features of the instance, which can be extracted with our tool.
Possible approach and technical challenges: The main idea would be to define a series of features for instances, which are numerical attributes associated with that instance (number of variables, some complexity measures, etc.), as well as what makes a solver the best for an instance (here, correctness and efficiency are clearly among the main aspects which should be factored in). Then, we should split existing instances into training- and test-sets. Based on the numerical attributes (features) from training-instances (extracted using database-functions) and the solving information stored in database, we can detect the best solver (w.r.t. correctness and efficiency) for each instance, and use this information as label for that instance. Label in a similar way the test-data. We then apply one/several learning-algorithm/s to the training data, and test the resulting algorithm on the test-set, to see how reliable it is. If reliable enough, the model can be used to predict which solver is likely to perform best on new instances.
Novelty: Tools exist already which select a solver according to the properties of an instance [MBK$^+$21]. However, this selection is deterministic and somehow superficial. This approach, based on our tool, would be the first to produce results of the type "Use solver X to solve this instance", based on a deep apriori analysis of the existing benchmarks and the performance of existing solvers on these benchmarks. In a sketch-implementation of this approach, in which we used some very basic features of the instances to define their attributes, and implemented neural-networks and decision-trees as basic learning algorithms, we obtained the following results. Both these algorithms performed similarly on the test-set. The fact that different benchmarks contain structurally different classes of constraints, affected the performance, but this stayed relatively high: it varied from 70% to 90% accuracy. Some improvement ideas would be to do a better feature-selection, as well as use more advanced learning-algorithms.

A.3 qlang predicates, functions, and extractors

The tool is currently under heavily development. Stated today we offer the following predicates, functions and extractors which will be extended continuously.

We plan to offer a shared platform to exchange custom implementations of the aforementioned tools.

In our implementation of BASC, a predicate is based on an interface in `smtquery.smtcon.exprfun` which corresponds to collecting data for the newly defined predicate. After providing a name and a version number, the user implements an `apply`- and a `merge`-function as mentioned before. The `apply`-function receives an AST expression and a pointer to previously calculated data and performs requested modifications to the data. Since the `apply` function computes the information bottom-up within our AST, the user also provides a neutral element for this computation, which might be an empty dictionary, an empty list, or simply an integer. The interfaces also requires the implementation of a `merge`-function which aims to combine the data received from children-expressions within the AST in a node. Once this information gathering interface is implemented, we register the application within `smtquery.intel.plugin.probes.intels` by providing a unique identifier which points to a tuple consisting of the function-implementations and the neutral element. Further, we register our predicate at `smtquery.intel.plugin.probes.predicates` providing a unique name and the predicate. Afterwards, the name is directly usable within our query language.

The `apply`-function, primarily allowing modifications of an AST, requires the implementation of a base-class defined within `smtquery.apply`. We provide a name and the expected behaviour, making sure to return an internal SMT-LIB object. After the successful implementation, we register this new class within our `PullExtractor` by providing its name. Again, afterwards, the `apply`-function is immediately usable within the query language.

The extractor allows exporting potentially modified benchmarks in an own format. BASC allows either printing the converted data directly to the terminal or redirecting it to a file using `smtquery.ui.Outputter`. To name a few examples, we might want to translate the benchmarks into a different format, export some plot, or obtain a modified SMT-LIB instance. To implement an extractor, we proceed similarly to the previously seen `apply`-function and implement a simple class within `smtquery.extract`. We provide a name and a function preforming the export based on our AST using the `Outputter`. We again register our new extractor to the `PullExtractor` by simply providing its class name, allowing us to use it in the query language. Currently, all data exported by our extractors is stored in a seperate folder in `output` located in the root of BASC.

A.4 Using BASC

In this section, we explain the basic commands of BASC and showcase some applications of BASC ranging from simple to more sophisticated experiments.

BASC provides a single executable located at `bin/smtquery` allowing to access all features of our toolbox by positional arguments. We run BASC by executing `python3 bin/smtquery` in the root folder of the project. In the following, we list the key arguments while more arguments are explained using the help command.

1. `initdb`: initializes a fresh database containing all instances stored in the file system at `data/smtfiles`.
2. `updateResults`: runs all available SMT-solvers on all registered benchmarks and stores the obtained results.
3. `allocateNew`: iterates through the file system and links new benchmark set within the database.
4. `qlang`: invokes an interface to pose queries using QLANG.
5. `smtsolver`: runs an smt-solver on a particular instance, e.g. `smtsolver CVC4 woorpje track01 01_track_1.smt` runs CVC4 on instance `01_track_1.smt` of track `track01` of the `woorpje` benchmark set.

Next, we show some examples of benchmark analysis, realizable by our tool.

§ *A variable dependency analysis.* To speed up the solving process for a particular string constraint, one might be interested in splitting a formula into multiple, independent sub-formulae. A relatively naïve way of splitting a formula is to determine whether the parts of the input formula in which each variable occurs, and see how they overlap. We can use BASC to visualize the interactions between variables within a formula, using the AST data structure. Whenever an AST is built starting from an SMT-LIB file, we store the variable occurrences for each node, which is actually the only information we need to build a simple variable-dependency graph. So, we have to define the generation of the corresponding plot by implementing the aforementioned class interface in `smtquery.extract`, call it `VarDepPlot`, and register it, after implementing the required logic in the `PullExtractor`. Now we can simply ask a query, e.g. **Extract** `VarDepPlot` **From** `*` **Where** `hasWEQ`, creating a variable-dependency plot for each registered instance which contains at least a single word equation.

§ *Analyzing the performance of a string solver.* To review the performance of a particular solver, one is usually interested in getting a comparison w.r.t. other solvers. BASC allows exporting a summary table and the commonly used cactus plot to compare string solvers on benchmarks. For instance, we want to see whether CVC4 is performing well enough on the WOORPJE benchmark set. First, we need to trigger CVC4 on the respective benchmark by executing, e.g., **Select** `Name` **From** `woorpje where isSAT(CVC4)`. Then, we can obtain a summary table by posing the query **Extract** `ResultsTable` **From** `woorpje` and a cactus plot by simply changing the extractor asking the query **Extract** `CactusPlot` **From** `woorpje`.

§ *Modifying Instances.* Analyzing the real-world benchmarks w.r.t. to a particular type of constraints can be achieved by simply neglecting all others. For example, to analyze the structure of the occurring word equations, one may simply pose the query **Extract** `SMTLib` **From** `*` **Where** `hasWEQ` **Apply** `Restrict2WEQ` to obtain cleaned SMT-LIB files. Naturally, also in this case, we can use any extraction function implemented to acquire the data which we are interested in.

§ *Finding and analyzing sub-theories.* In [BDG+21] we have analyzed a large set of benchmarks w.r.t. regular expression membership queries. This kind of queries plays a central role in verifying security policies, by allowing to restrict the set of possible input strings by a regular language [BKM+21]. The inspection of the respective benchmarks was performed using a sequence of different handcrafted scripts, restricted to a particular use case. BASC provides the means to easily extract this data by simply defining predicates analyzing the regular languages (i.e., regular expressions) occurring in the benchmarks. For example, to gather all instances solely containing regular membership constraints asking whether a string without variables or a single variable is a member of a regular expression without complement or intersection is achieved by posing the query **Select** `Name` **From** * **Where** `isSimpleRegex`. The key difference is that the definition of the particular predicate is much simpler, due to the extendable structure of BASC. As such, we can now simply combine the acquired information with newly developed predicates. Since this analysis lead to a well performing algorithm, presented in [BKM+21], we are optimistic that our tool can be used to extract such relevant data, ultimately leading to better techniques in the area of solving string constraints.

§ *Analyzing the structure of instances.* BASC also offers the possibility of a more in-depth analysis of the (syntactic-)structure of the instance. For instance, knowing that all string variables occurring in a formula are subject to constant-length upper bounds allows us to rephrase the problem as a constraint satisfiability problem over finite domains, and ultimately may lead to faster solutions for it. To extract a list of instances having only length-upper-bounded variables, we can pose the query **Select** `Name` **From** * **Where** `isUpperBounded`. The predicate analyzes the syntax of the constraints and extracts relevant information. Another interesting aspect, which can potentially lead to a better choice of an algorithm for solving a particular instance, is the analysis of its combinatorial structure. For example, if we know that each variable is occurring at most twice inside a formula, or that each word equation is of the form $\mathsf{x} \doteq \alpha$ where x is a variable not occurring in α nor anywhere else in the formula, we can use customized solving techniques, and solve the instances more efficiently. Obtaining such information is done by using the predicate `isQuadratic`, resp. `isPatternMatching`.

§ *Efficiency.* All operations implemented so far in BASC have a reasonable run-time. Clearly, it is inherent that some operations require a rather long execution time: firstly, we analyze a huge set of instances, and, secondly, the analysis we apply might involve complicated predicates, which are provably computationally hard. Our approach to speeding-up this process is to allow the incremental inclusion in the stored-data of the results of various queries, on which one can build efficiently more complex queries. Summing up, our goal was to build a tool allowing information extraction from benchmarks of string constraints, using a state of the art home-computer, without having to go deep in implementation details.

Let us give some precise examples. For instance, we have recorded runtimes (on a standard PC) of 19807 instances of the Kaluza-set, and obtained the following results: database initialization (21s), obtaining results from CVC4, Z3STR3, Z3SEQ (612s), using several apply-functions without caching (198s) and with caching (32s), extracting plots (200s-300s). The runtimes of the apply-functions are similar due to the underlying data-structure, which efficiently stores/provides useful information.

More details and examples regarding runtimes are given on the website of BASC `https://b4sc.github.io/`.

Predicates To be used within the **Where** part of the query. All predicates can be combined using the common logic connectives, e.g. and, or, and not.

`hasWEQ`: filters to all instances which contain word equations.
`hasLinears`: filters to all instances which contain linear length constraints.
`hasRegex`: filters to all instances which contain regular membership predicates.
`isSimpleRegex`: filters to all instances which are of the simple regular expression fragment (see [BDG+21]).
`isSimpleRegexConcatenation`: filters to all instances which are of the simple regular expression fragment with concatenation (see [BDG+21]).
`isUpperBounded`: filters to all instances where the syntax of the formula allows obtaining a length upper bound for each string variable.
`isQuadratic`: filters to all instances where each string variable is occurring at most twice.
`isPatternMatching`: filters to all instances which only contain word equations of the kind $\mathsf{x} \doteq \alpha$ where x is a variable not occurring anywhere else in the present formula and α is a string (potentially containing variables other than x).
`hasAtLeast5Variables`: filters to all instances containing a least 5 string variables.
`isSAT(`s`)`: filters all instances where $s \in \{\text{CVC4}, \text{Z3Str3}, \text{Z3Seq}\}$ declared satisfiable.
`isUNSAT(`s`)`: filters all instances where $s \in \{\text{CVC4}, \text{Z3Str3}, \text{Z3Seq}\}$ declared unsatisfiable.
`hasValidModel(s)`: filters all instances where $s \in \{\text{CVC4}, \text{Z3Str3}, \text{Z3Seq}\}$ returned SAT with a valid model.
`isCorrect(s)`: filters all instances where $s \in \{\text{CVC4}, \text{Z3Str3}, \text{Z3Seq}\}$ returned SAT with a valid model or UNSAT was returned by the majority of used solvers.
`isFaster(s1,s2)`: filters all instances where $s1, s2 \in \{\text{CVC4}, \text{Z3Str3}, \text{Z3Seq}\}$ and $s1$ determined some result quicker than $s2$.

Functions To be used within the **Apply** part of the query.

`Restrict2WEQ`: removes all other predicates than word equations.
`Restrict2Length`: removes all other predicates than linear length constraints.
`Restrict2RegEx`: removes all other predicates than regular expression membership queries.
`RenameVariables`: renames all variables to a standard format (i.e. `str01, int01`).
`DisjoinConstraints`: splits and-concatenated boolean constraints into separate assertions.
`SortConstraints`: lexicographically sorts all constraints.
`ReduceNegations`: shortens sequences of not, keeping the original polarity.
`EqualsTrue`: simplifies constraints comparing boolean expressions to true.

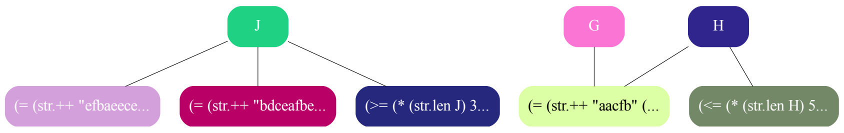


Fig. 3. Example variable dependency plot.

Extractors To be used within the **Extract** part of the query.

`MatchingPie`: exports result as a pie chart.
`CactusPlot`: export result as a cactus plot.
`SMTPlot`: exports the instances visualized as tree diagram.
`VarDepPlot`: exports the dependency plots of all instances.
`ResultsTable`: prints the results in terminal.
`SMTLib`: exports the resulting instances as SMT-LIB files.
`Count`: prints matching instances count and distribution.
`InstanceTable`: prints the matching instances and solver's results.

A.5 Example outputs of extractors

We discuss here some capabilities of our extractors which, unfortunately, were only briefly discussed or have not been included at all into the main part of the paper due to the space constraints. We will ensure the availability of more examples in our manual. Note that the results reported in these example queries (e.g., running times) are depending on the machine on which they are performed.

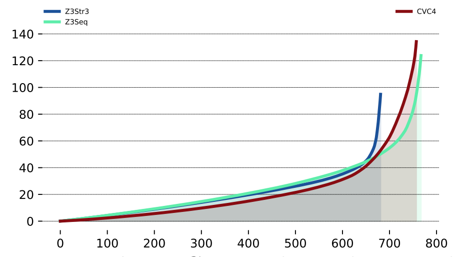
§ *A variable dependency analysis.* Posing the query of the aforementioned example might, e.g., have the instance of Listing 1.2 leading to the plot of Figure 3 where variables G, H and J are top nodes and the corresponding assertions are to bottom nodes.

```
(set-logic QF_S)
(declare-fun H () String)
(declare-fun G () String)
(declare-fun J () String)
(assert (= (str.++  "aacfb" G "abdeddaaa")  (str.++  "aacfbdffebaaaaac" H "aaa"
     ↪ ) ))
(assert (= (str.++  "efbaeecedaaecfceffaffaedfcebcf" J "aeaadcbe")  (str.++  "e
     ↪ " J "aeecedaaecfceffaffaedfcebcf" J "aeaadcbe") ))
(assert (= (str.++  "bdceafbededddcfcacffdeaefcfa" J "dbabcdebee")  (str.++  "
     ↪ bdceafbededddcfcacffdeaefcfa" J "dbabcdebee") ))
(check-sat)
```

Listing 1.2. Instance for variable dependency plot

The edges in the figure indicate the presence of a variable, allowing us to split the instance accordingly.

§ *Analyzing a string solvers performance.* The results of the query **Extract** `ResultsTable` **From** `woorpje` are visualized by BASC as shown in Figure 4.



```
                      Z3Str3    Z3Seq     CVC4
----------------    --------  -------   -------
SAT                 512       604       594
UNSAT               170       164       164
Unknown              13         0         0
Timeout             114        41        51
Crash                 0         0         0
Time w/o Timeout     97.5154  123.816   134.242
Total Time          669.859   329.158   389.631
```

Fig. 4. Cactus plot and terminal results for an example query.

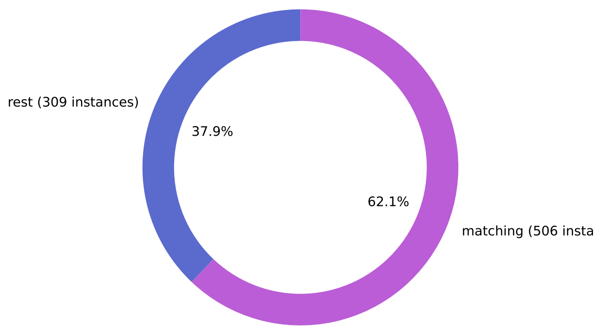


Fig. 5. Example for a matching pie plot.

Another interesting visualization can be exported using the `MatchingPie` . We might want, for example, to visualize how many instances have been answered satisfiable by both CVC4 and Z3Str3 within the **`woorpje`** benchmark set. To obtain Figure 5 we simply pose the query **Extract** `MatchingPie` **From** `woorpje` **Where** `(isSAT(CVC4)`**and** `isSAT(Z3Str3))` revealing that the solvers jointly answered `SAT` on 506 out of 815 instances. More interestingly, we can also ask for disagreeing answers with respect to solvers returning `SAT` by changing the query to **Extract** `MatchingPie` **From** `woorpje` **Where** `((isSAT(CVC4)`**and not** `isSAT(Z3Str3))`**or** `(`**not** `isSAT(CVC4)`**and** `isSAT(Z3Str3)))`.

§ *Manual inspection of instances.* To identify potential new predicates, functions, and extractors, it might be helpful to observer an SMT-LIB instances manually. Since it is hard to grasp the structure especially for large instances, BASC offers an export of instances into a tree-like plot making inspection way easier. To export a plot as seen in Figure 6 for the instances printed in Listing 1.1 we simply pose the query **Extract** `SMTPlot` **From** $*$ which generates plots for all registered instances. We highlight different occurrences of the same expressions in the same color to easily observe their distribution.

§ *Counting instances with particular properties.* To count how many instances containing regex-membership predicates are in a particular benchmark (e.g. for JOACO-Suite) we can pose the query **Extract** `Count` **From** `joacosuite` **Where** `hasRegex`.

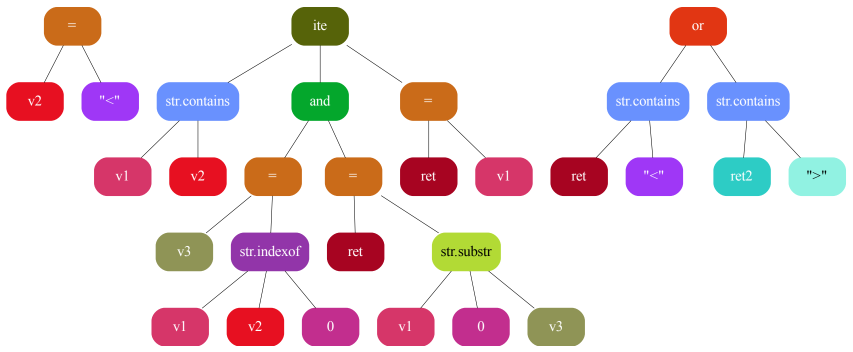


Fig. 6. Tree-like Graph for an example SMT-LIB file.

References

[AAC+15] 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 *International conference on Computer Aided Verification*, pages 462–469. Springer, 2015.

[ABB15] Abdulbaki Aydin, Lucas Bang, and Tevfik Bultan. Automata-based model counting for string constraints. In *Computer Aided Verification: 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, pages 255–272. Springer, 2015.

[Ama21] Roberto Amadini. A survey on string constraint solving. *ACM Computing Surveys (CSUR)*, 55(1):1–38, 2021.

[BCD+11] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *CAV 2011*, pages 171–177. Springer, 2011.

[BDG+21] 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 *WORDS 2021*, pages 50–64. Springer, 2021.

[BKM+21] 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 *CAV 2021*, pages 289–312. Springer, 2021.

[BST+10] Clark Barrett, Aaron Stump, Cesare Tinelli, et al. The SMT-lib standard: Version 2.0. In *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England)*, volume 13, page 14, 2010.

[BST+15] Clark Barrett, Aaron Stump, Cesare Tinelli, et al. The SMT-lib standard: Version 2.5. *Available at:*, https://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.5-r2015-06-28.pdf, 2015.

[CCH+18] Taolue Chen, Yan Chen, Matthew Hague, Anthony W. Lin, and Zhilin Wu. What is decidable about string constraints with the replaceall function. *Proc. ACM Program. Lang.*, 2(POPL):3:1–3:29, 2018.

[CFH+22] Taolue Chen, Alejandro Flores-Lamas, Matthew Hague, Zhilei Han, Denghang Hu, Shuanglong Kan, Anthony Widjaja Lin, Philipp Rümmer, and

Zhilin Wu. Solving string constraints with regex-dependent functions through transducers with priorities and variables. *CoRR*, abs/2111.04298, 2021, to appear in POPL 2022.

[CHL$^+$19] 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.

[DEK$^+$19] Joel D. Day, Thorsten Ehlers, Mitja Kulczynski, Florin Manea, Dirk Nowotka, and Danny Bøgsted Poulsen. On solving word equations using SAT. In *International Conference on Reachability Problems*, pages 93–106. Springer, 2019.

[DKM$^+$20] Joel D. Day, Mitja Kulczynski, Florin Manea, Dirk Nowotka, and Danny Bøgsted Poulsen. Rule-based word equation solving. In *Proceedings of the 8th International Conference on Formal Methods in Software Engineering*, pages 87–97, 2020.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS 2008*, pages 337–340. Springer, 2008.

[Hag19] Matthew Hague. Strings at MOSCA. *ACM SIGLOG News*, 6(4):4–22, 2019.

[KGG$^+$09] Adam Kiezun, Vijay Ganesh, Philip J Guo, Pieter Hooimeijer, and Michael D Ernst. HAMPI: a solver for string constraints. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 105–116. ACM, 2009.

[KLRS22] Shuanglong Kan, Anthony W. Lin, Philipp Rümmer, and Micha Schrader. Certistr: A certified string solver (technical report). *CoRR*, abs/2112.06039, 2021, to appear in CPP 2022.

[KMNP21] Mitja Kulczynski, Florin Manea, Dirk Nowotka, and Danny Bøgsted Poulsen. ZaligVinder: A generic test framework for string solvers. *J. Software: Evolution and Process*, page e2400, 2021.

[LH18] Quang Loc Le and Mengda He. A decision procedure for string logic with quadratic equations, regular expressions and length constraints. In *APLAS*, volume 11275 of *Lecture Notes in Computer Science*, pages 350–372. Springer, 2018.

[MBK$^+$21] Federico Mora, Murphy Berzish, Mitja Kulczynski, Dirk Nowotka, and Vijay Ganesh. Z3str4: A Multi-armed String Solver. In *FM 2021*, pages 389–406. Springer, 2021.

[The] The SMT-LIB Initiative. The SMT Library. `https://smtlib.cs.uiowa.edu/benchmarks.shtml`. Accessed: 2022-01-17.

[YAB10] Fang Yu, Muath Alkhalaf, and Tevfik Bultan. STRANGER: An Automata-based String Analysis Tool for PHP. In *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'10, pages 154–157, Berlin, Heidelberg, 2010. Springer-Verlag.

[ZZG13] Yunhui Zheng, Xiangyu Zhang, and Vijay Ganesh. Z3-str: A z3-based string solver for web application analysis. In *ESEC/SIGSOFT FSE 2013*, pages 114–124, 2013.