Mining and Analysing String Constraints

⟨Authors hidden due to double blind review⟩

Affiliation hidden due to double blind review

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 SMTQuery: an SMT-LIB benchmark analysis tool for string constraints. SMTQuery 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 a novel SQL-centred language, called qlang.

1 Introduction

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. Worth noting: in this specific case, the constraint satisfiability problems are over an infinite domain. Motivations for theoretical and practical investigations in this area come from the verification of security-related programming errors (e.g., detecting security flaws such as *SQL injection* and *cross-site scripting attacks*) or symbolic execution of string-heavy languages. Excellent overviews of the main definitions and fundamental results as well as of the many recent developments related to the theory and practice of string solving are provided by [2, 21].

Relevant to our work, on the practical side, a series of dedicated string constraint solvers were developed (see, e.g., Norn [1], Stranger [30], ABC [4], Woorpje [16, 18], OSTRICH [14], CertiStr [23]), but also well-established general-purpose SMT solvers (such as cvc5 [5] and Z3 [8, 19, 27]) 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 by Kulczynski et al. [25]. These benchmarks were extracted both from real-world and 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 set [32]. It models a conditional choice of a return string variable `ret` making assumptions on other variables having the data type String. For examples of systems of string constraints, as found in the benchmarks, see, e.g., [6, 29].

Moreover, there exists now a unified string-logic standard as part of SMT-LIB, and the tool ZaligVinder [25] brings together a set of relevant benchmarks and introduces a uniform benchmarking framework. Nevertheless, there are still some notable challenges related to string-solving benchmarks:

- the benchmarks are still largely unclassified w.r.t. satisfiability,
- a classification w.r.t. satisfiability is, thus, often based on older string solvers verdicts — existing string solvers thus become *oracles* for developers of new string solvers, but if the oracle has implementation errors then that error is inherited through future developments,
- the benchmarks mostly originate from automated tools, which means that they are not developed for human interpretation, making it even harder to validate models/unsatisfiable results from string solvers,
- the benchmarks are still mostly uncategorized w.r.t. the type of string constraints they contain, and solvers addressing specific types of constraints have to first preprocess the existing benchmarks and extract the relevant constraints [9, 13]. Lacking this kind of classification makes harder the development of algorithms targeting string constraints that (often) appear in practice, since there is no obvious way of knowing what constraints are common nor their structure in different areas.

Research Tasks. In this context, we formulate two main research tasks addressing the central issues related to curating and processing string-solving benchmarks.

1. Identify, store, and organize a comprehensive collection of benchmarks for string solving as a database, allowing querying, exporting, and data mining 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 easily compare their results
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 mining data 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 larger collections of more general benchmarks containing SMT or SAT instances [22, 29].

Our contribution. We propose SMTQuery: a benchmark analysis tool (accessible at `http://smtquery.github.io`) for string constraints. It is focused on benchmarks (and underlying theories) related to string solving, but can easily be extended to cover other theories. It thereby offers the foundations for a more comprehensive database containing insights of general SMT formulae. It should also be mentioned that although SMTQuery has benchmarking capabilities, the major new contribution of this paper is the data extraction system of SMT-LIB benchmarks and further processing mechanisms.

SMTQuery is implemented in Python 3 and offers a collection of analysis and data mining tools for the most comprehensive database of string benchmarks (collected from the literature and presented in SMT-LIB format), based on a novel SQL-centred language called qlang. Besides basic database management, benchmark querying, and analysis capabilities, SMTQuery also offers an interface for running and testing string solvers on the benchmarks. The results of such runs can then be collected, stored, further analysed, and correlated to other properties of the respective benchmarks (computed using SMTQuery database queries). As such, SMTQuery offers solutions to our two research tasks. SMTQuery also offers 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. The user base, architectural details, and use cases of SMTQuery are discussed in the rest of the paper and the documentation of the tool.

2 Potential Applications of SMTQuery and User Base

We begin with examples from the literature, where this tool could have been used. In particular, our goal in this section is to show that there is a demonstrable need for analyses of the properties of benchmarks containing string constraints.

In the following, we overview several cases where hand-crafted benchmarks and ad-hoc analyses were created and used. SMTQuery offers a more general and easier-to-use framework for such analyses.

The first example for an ad-hoc analysis is [12], which motivates the Straight-line fragment of string constraints by noting (as the product of the aforementioned analysis) that the Kaluza benchmark set falls within it. As the Kaluza benchmarks become older and new sets emerge, it would be beneficial to have similar updated analyses for these new sets of constraints, so that assumptions about practical instances do not become out-of-date. SMTQuery is aimed to make these much quicker and easier.

In [26] the authors argue that a new, hand-crafted benchmark, whose instances fulfil some specific properties, is required to be able to compare their procedure with existing solvers. This paper could have benefited from our tool. In such cases, it would be faster and more transparent to use SMTQuery to extract from existing benchmarks the instances exhibiting the addressed properties, and showcase the novel algorithms on already existing, provably relevant instances.

A similar argument can be made related to [13], where the authors say "*There are no standard string benchmarks involving RegExes[...]*". Using SMTQuery such benchmarks could be extracted from existing, provably relevant instances.

Nevertheless, in [9], a set of over 100000 benchmarks was analysed ad-hoc, to extract string constraints containing only regular expressions and linear arithmetic and detect their structural complexity, with the ultimate goal of producing an efficient solver for such constraints. Such an analysis is inherently complex and tedious. Howver, SMTQuery's abilities potentially simplifies this process.

Therefore, it seems that analyses like those from the aforementioned examples required significant effort, replicated every time a new analysis is needed. We expect SMTQuery to be used routinely by developers of string-solvers (like those mentioned in, e.g., [2, 21]) to provide faster and transparent evaluation, up-to-date context, and applicability evidence for their algorithms.

On the other hand, many theoreticians use string-solving to motivate their works. SMTQuery can provide real evidence supporting this and also inform future directions of study. In general, Hague [21] presents examples of research groups likely to use SMTQuery. Concretely, a prototype version of SMTQuery was already used in [17], in the context of combinatorial pattern matching, where the authors were interested in extracting and understanding the structure of regular expressions used in practice (see also Section 4).

In conclusion, the cases overviewed above, as well as examples found in literature, immediately reveal three different communities of potential users of SMTQuery: firstly, the community of string solver developers, for which it eases the performance-analysis of specific solvers, on specific benchmarks, and, thus, helps in discovering the strengths and weaknesses of each string solver, for instance by identifying certain features of the input and correlating them to the solver's performance. Secondly, the theory community: SMTQuery facilitates the further understanding of structural properties of specific classes of word equations, relevant in practice, which can be the focus of theoretical investiga-

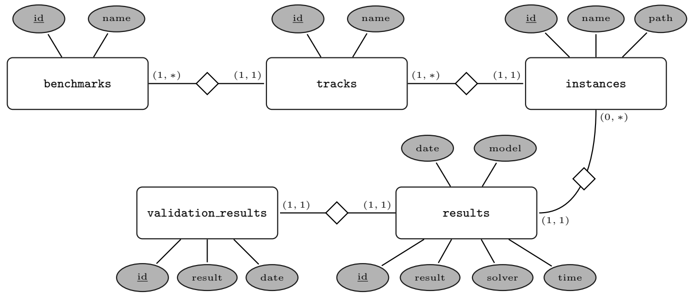


Figure 1: SMTQuery's SQLite database schema

tions. Finally, the end-users, not explicitly mentioned before: 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 SMTQuery, and use the offered methods to analyse it.

3 Architecture of SMTQuery

We begin the technical part of this paper by discussing the main ideas behind the architecture of SMTQuery.

SMTQuery provides a series of mechanisms easing the access to a comprehensive set of benchmarks, based on an SQL-inspired query language called qlang. The tool is built such that it can be run on an everyday workstation within a terminal and it aims to provide answers to the user's questions regardless of the time it takes to get them. To this extent, we have tried to make SMTQuery as flexible as possible, giving easy-to-use entry-points to adding custom algorithms for the analysis of string solvers or benchmarks without requiring high-performance servers. Nevertheless, due to multiprocessing, we allow running our tool in a server environment which speeds up the answering of the asked questions, providing rather superb response times, as we discuss in Section 4. The input is given in our novel query language called qlang, which allows accessing and analysing benchmarks following the SMT-LIB standard regardless of their origin, directly in a terminal prompt. 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 `benchmarks`, `tracks`, `instances`, `results`, and `validation_results` visualised in Figure 1. 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 the tables `benchmarks` 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. Initially, a

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

Figure 2: Syntax of QLANG.

new benchmark set including its instances is always registered in our database. 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 and the conclusions are stored in table `validation_results`.

We provide an easy interface, allowing us to add additional solvers. Currently, SMTQuery implements it for cvc5, 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. We took the scheduling engine allowing multiprocess runs of solvers, as well as the runners developed for the benchmark framework. This includes special handling for different string-solvers as explained in [25]. Furthermore, we reuse the cross-validation mechanism: ZaligVinder runs all competing solvers and whenever a server returns `SAT`, we check the validity by asserting the model into the original instance and using another solver to check correctness. In the case of `UNSAT` and when no other solver returned a valid model, we use a majority vote upon all solvers' results.

To allow gathering new insights about the benchmarks, SMTQuery offers an interface permitting the definition of custom benchmark-analysis predicates, which can directly interact with the SMT-LIB instances and the pre-calculated 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 we use.

The language QLANG. Let us now go a bit more into details regarding our query language QLANG. As its 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 2.

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, SMTQuery implements several default predicates. For instance, the default predicate `hasWEQ` selects the benchmarks containing at least one word equation, or `isSAT(`*solver*`)` returns instances being declared satisfiable by the particular *solver*. Worth noting, our interface allows also defining custom predicates. As far as the implementation is concerned, when evaluating a predicate (as, for example, our default predicates) 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 (just as we did for the default predicates) 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 related information attached to each node is called IntelDictionary which will be explained in detail within the next paragraph. We visualise this procedure in Figure 3. As a basic, yet illustrative example, consider the word equation $a\mathsf{Y}ab\mathsf{X} \doteq \mathsf{Z}abb\mathsf{Y}$ where a and b are terminals and X, Y and Z are variables. The calculation of the number of occurrences of each variable in a node starts by counting these occurrences within the two sides of the equation leading to the sets $\{ (\mathsf{X}, 1), (\mathsf{Y}, 1) \}$ and $\{ (\mathsf{Y}, 1), (\mathsf{Z}, 1) \}$. Our root node corresponds to the equality $\doteq$. We apply the merge-function, which adds up the occurrences of variables in the children nodes of the current node, to obtain the set $\{ (\mathsf{X}, 1), (\mathsf{Y}, 2), (\mathsf{Z}, 1) \}$ which indeed is the desired data for the root node, since $\doteq$ does not contain any other variables. In general, 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. Continuing our previous example, asking whether a word equation is quadratic via the predicate `isQuadratic` can simply use the previously calculated data (number of occurrences of the variables) and simply return `true` if and only if each variable has at most two occurrences.

Finally, 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 for **Select** queries. Additionally, while executing a **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 or simply restricting an instance to a particular kind of string constraint

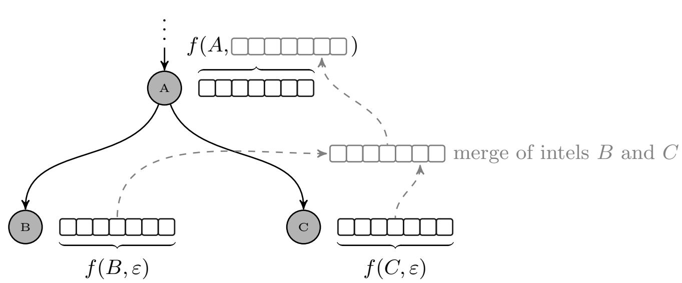


Figure 3: Calculation of the IntelDictionary in our AST. f is the `apply`-function, ε the corresponding neutral element, and A, B, C arbitrary nodes and their IntelDictionary (boxes next to nodes) in our AST corresponding to an SMT-LIB instruction.

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

Figure 4: Internal representation of string constraints.

(e.g. word equations). Moreover, this interface allows the application of external procedures on the whole AST, such as applying a fuzzer like StringFuzz [11] 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 structure. As a simple example, to count instances which exhibit certain properties, one could use the predefined operation `Count`. 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 a query, to obtain a list of all benchmarks containing word equations and determined satisfiable by the string solver cvc5, we can execute the query **Select** `Name` **From** `*` **Where** `(hasWEQ` **and** `isSAT(CVC5))`. A second example is removing all other constraints than word equations from our benchmarks and exporting the resulting SMT-LIB files. We use an **Extract** query and pose **Extract** `SMTLib` **From** `*` **Where** `hasWEQ` **Apply** `Restrict2WEQ`[1].

The AST structure. At the core of our implementation is the AST data structure, which is directly derived from the SMT-LIB instances. A string constraint defined in SMT-LIB contains variable declarations and (potentially) multiple asserts of formulae being based on string constraints connected by the common connectives (note that the string constraints are not quantified in the respective standard). All asserted formulae have to be satisfied at the same time. Based on this structure, our AST is a parse tree derived according to the formal grammar from Figure 4, modelling each formula. As it can be seen there, SMTQuery currently supports common string-constraints: word equations, linear arithmetic-over-lengths, regular-language-membership, Boolean constraints. We use Z3 as input parser for SMT-LIB, so SMTQuery parses all constraints Z3 handles. Our internal representation uses a generic expression to represent constraints/types not covered explicitly yet in our grammar. Thus, as already hinted at the end of the Introduction, SMTQuery can be canonically extended to address other constraint types and, as such, other theories.

Let us now go into some of the technical details on which the parsing process and the ASTs are based. We begin by explaining the grammar of Figure 4, which lays the foundations for the internal representation of string constraints. In this grammar, each expression *Expr* (which corresponds to an SMT-LIB formula) 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 expres-

[1] A list of the available options is printed when executing our tool and available in SMTQuery's documentation.

sion 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 [28] 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. This allows quickly re-accessing of stored information, since, e.g. for selecting all instances containing word equations, only the root node's IntelDictionary has to be checked when using the predicate `hasWEQ`.

One key aspect of SMTQuery'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.

Summary. In Figure 5 we overview the overall architecture of SMTQuery. A user poses a qlang query q to the command line interface. After parsing the query q the core logic acquires relevant information about the selected benchmarks and schedules solver runs at any time needed. The query q either has the form **Select** f_s **From** d **Where** c or **Extract** f_e **From** d **Where** c **Apply** f as opposed in the grammar shown in Figure 2. For the selection of the benchmarks d, we query our SQLite database which returns related information (i.e. a pointer to the AST, a unique id, and file-system path) for each requested instance. We apply the predicate c to each instance obtained from the previous query, potentially removing it from our selection. The predicate c makes use of the IntelDictionary stored in our ASTs. When the AST is not available, the Z3's output of the parsed SMT-File is translated into our AST. Furthermore, if parts of the requested data in the IntelDictionary are not available, we recalculate it on the

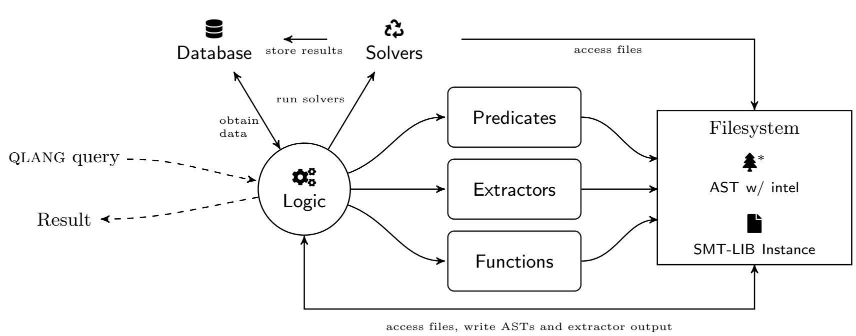


Figure 5: Architectural overview of SMTQuery.

fly and additionally enrich our AST with the newly obtained information. We do not store the IntelDictionary within our SQLite database to stay as flexible as possible. Since each node of our AST enriches the IntelDictionary of its children, storing this information inside our database would result in storing a link to each node. Secondly, adding new entries to the IntelDictionary would require a modification of our database schema. A predicate c might also ask for solver related information (e.g. `isSAT(Z3Str3)`, asking for all instances declared satisfiable by Z3Str3). In this case, we again query our SQLite database for the requested information. Whenever the data is not available, we automatically call the solver and store the corresponding results within the database, making it accessible for further queries. The same step calls the verification mechanism explained earlier. Posing a **Select** query to SMTQuery outputs the results being specified by f_s directly into the user's terminal. An **Extract** query acts differently depending on the extractor f_e. As explained previously, an extractor can modify matching instances based on its own needs. Therefore, f_e might write data to the file-system (e.g. a cactus plot using `CactusPlot` or an SMT-File using `SMTLib`) or simply print a result to the user's terminal (e.g. a summary of solver results using `InstanceTable`). If the query contains a function f within it **Apply**-part, the core logic performs modifications according to the specification given by f before using the extractor f_e.

The operations implemented so far in SMTQuery heavily differ in their run time. Clearly, it is inherent that some operations require a rather long execution time: firstly, we analyse 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 into implementation details. In the next second, we report the running times of executed queries.

For more detalis, see SMTQuery's site `http://smtquery.github.io`.[2]

4 Use cases and examples

This section is devoted to examples of problems which can be addressed with SMTQuery. We pose a task and describe the difficulties arising while solving it. Afterwards, we show how certain problems can be addressed using SMTQuery and support our approach with results and statics based on ZaligVinder's benchmark set.

Our experimental setup is built upon 114468 different SMT-LIB instances gathered in [25] containing 19 different sets mainly stemming from real-world applications and solver developers as explained in our introduction. Firstly, to

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

SMT-LIB 2.5 keyword	SMT-LIB 2.6 keyword
`int.to.str`	`str.from_int`
`str.to.int`	`str.to_int`
`str.in.re`	`str.in_re`
`str.to.re`	`str.to_re`
`re.nostr`	`re.none`
`re.empty`	`re.none`
`\x0`n	`\u{`n`}`
`\x`m	`\u{`m`}`

Table 1: Translation from SMT-LIB 2.5 to 2.6 for $n \in \mathbb{N}_{\leq 9}$ and $m \in \mathbb{N}_{>9}$

incorporate the most recent release of cvc5, we manually translated these benchmarks into SMT-LIB 2.6. The gathered instances were still in SMT-LIB 2.5 format which is no longer supported by cvc5. The translation itself is a straightforward renaming of the keywords and functions given in Table 1. We set up SMTQuery on a server running Ubuntu 18.04.4 LTS with two AMD EPYC 7742 processors having a total of 128 cores and 2TB of memory. We integrated cvc5's version 1.0.1 and Z3's version 4.10.1 binaries from their official sources.

Before we consider actual use cases, we use SMTQuery to get a better intuition on the used benchmarks and obtain some insights. First, we initialise the SQLite database such that it contains the schema shown in Figure 1 and links all of our 114468 instances accordingly. This process took 4.44 minutes. We now use our built-in predicates to observe that 80284 instances contain word equations (using the predicate `hasWEQ`), 57257 contain regular-expression membership constraints, and 59763 contain linear arithmetic over string length. Additionally, we discovered that 30393 contain higher-order functions (e.g. `str.substring`, `str.replace`). The running time for each query without using the cache was about 7.16 minutes. Using our pre-cached ASTs allows us to acquire the above values in roughly 70 seconds.

The above values do not require satisfiability results of the embedded string solvers. As mentioned in the previous section, all integrated solvers will be run automatically. To quickly access cached results, our tool allows running all solvers (including verification) in advance. The running times heavily depend on selected timeouts and machine power, as well as the performance of the embedded solvers (e.g. obtaining results takes longer if a solver times out more often). To give an intuition on the running times using previously stored results, we discover that cvc5 declares 71540 instances satisfiable. We obtain this value in 5.13 minutes.

We now move on to particular use cases. The first generic problem we address is the following:

Problem I: *Given a syntactically restricted subset of string constraints, determine instances belonging to this subset, and their distribution in benchmarks.*

Many theory papers provide insightful results (i.e., algorithms, complexity bounds, information about solution sets) for subclasses of string constraints [21]. Such subclasses include those defined directly (e.g., constraints in solved-form, acyclic or straight-line constraints) as well as indirectly (e.g., quadratic or regular word equations). Knowing how applicable such results resp. 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, SMTQuery is valuable for researchers approaching subcases of string constraints, 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, in order to make it more applicable while preserving theoretical properties. This approach is also of value to researchers developing string solvers who will benefit from knowing which theoretical insights are most likely to be effective over a broad range of use-cases and which properties to target with their own optimizations and innovations.

When dealing with the aforementioned problem it is worth noting that, 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 or even constantly while solving them. Obtaining realistic data, therefore, requires taking into account the effects of rewriting processes concerning syntactic subsets.

Currently, there are no tools capable of properly addressing this problem and these challenges. Without SMTQuery, 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. In this context, it does not come as a surprise to see that such analyses have not been yet carried out even for major subclasses of string constraints.

We can give two concrete cases of the problem stated in this generic example. Firstly, we investigate the distribution of quadratic equations (a class of word equations, which can be solved using a technique inspired by Levi's lemma [15]) in the benchmarks. An analysis might result in the following questions:

1. For each benchmark, determine all instances consisting of quadratic equations only: **Select** `Name` **From** * **Where** `isQuadratic`. In 63 seconds using our cached AST SMTQuery outputs a list of all 47796 matching instances.
2. Count how many such instances are in each benchmark, and compute the ratio between the number of quadratic instances and the overall number of instances in each benchmark (e.g. for JOACO-Suite):

   Extract `Count` **From** `joacosuite` **Where** `isQuadratic`.

   After less than 1 second SMTQuery reports "Total matching instances: 51 of ↪ 94 within the selected set (54.25%)".

Secondly, motivated by the work performed in [7, 9], we want to determine all instances containing regular-membership predicates, and their distribution within benchmarks.

1. For each benchmark, determine all instances containing at least one regular-membership predicate: **Select** `Name` **From** * **Where** `hasRegex`. After roughly 70 seconds SMTQuery prints a list of 57257 containing regular-membership predicates.
2. Count how many such instances are in each benchmark, and compute the ratio between the number of instances containing regex-membership predicates and the overall number of instances in each benchmark (e.g. for JOACO-Suite):

   Extract `Count` **From** `joacosuite` **Where** `hasRegex`.

   After less than 1 second we obtain the output "`Total matching instances: 76 ↪ of 94 within the selected set (80.85%)`".
3. We are interested in gathering knowledge about how many of the instances containing regex-membership predicates fall into the PSPACE-complete fragment of simple regex-membership predicates (i.e. predicates of the form $\mathsf{x} \mathrel{\dot\in} R$, where x is a variable and R is a regular expression not containing complements). We pose:

   Extract `Count` **From** * **Where** `isSimpleRegex`.

   SMTQuery returns "`Total matching instances: 24486 of 114468 within the selected ↪ set (21.39%).`" in 2.10 minutes.

We move on to a second generic problem.

Problem II: *For a given string solver, understand the properties of instances on which it performs particularly well, and on which it performs poorly.*

Having insights are valuable for designing new and improving or optimising existing string solvers. It is also valuable for constructing *portfolio solvers* who simply choose a well-performing algorithm for a particular case (see e.g. [27]).

Clearly, some challenges stem from the same issues discussed in the previous example. Moreover, once we computed a set of instances on which a solver performs well and a set of instances on which that solver performs poorly, we need a reliable analysis tool to find properties which separate the two sets.

Tools already exist which assist the comparison of string solvers in terms of directly evaluating how they perform over a set or sets of benchmarks (e.g., [10,25]). This is sufficient for producing evidence of their effectiveness within the current landscape of solvers. However, without SMTQuery, it is difficult even to understand the character of particular benchmarks beyond very superficial observations. Thus, existing tools do not provide insights 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 a well-motivated criterion obtained by using SMTQuery.

We can give a concrete case related to the problem stated in this example. We would be interested in finding the set C of all the instances on which cvc5

```
Waiting for results ...
Instance                     Result CVC5      Time CVC5   Result Z3Seq       Time
     ↪ Z3Seq   Result Z3Str3      Time Z3Str3
------------------------  ------------  ----------  ------------
     ↪ ----------  ------------  ----------
pisa:pisa:pisa-011.smt2   Satisfied        0.00897606  Satisfied
     ↪ 0.0259043   Satisfied           0.0344819
pisa:pisa:pisa-009.smt2   Satisfied        0.0191097   Satisfied
     ↪ 0.0276228   Satisfied           0.028013
pisa:pisa:pisa-010.smt2   Satisfied        0.0167181   Satisfied
     ↪ 0.0258694   Satisfied           0.0266274
pisa:pisa:pisa-002.smt2   Satisfied        0.0235912   Satisfied
     ↪ 0.116755    Satisfied           0.0386019
pisa:pisa:pisa-000.smt2   Satisfied        0.0695572   Satisfied
     ↪ 0.0426866   Satisfied           0.0492182
...
```

Figure 6: Cut terminal output for the query posed in Problem II

provides a correct answer and Z3Str3 either provides a wrong answer or is slower in providing the correct answer and the instances the set Z of all the instances on which Z3Str3 provides a correct answer and cvc5 either provides a wrong answer or is slower in providing the correct answer. Then, for each of these sets, detect the number (and distribution) of instances containing regular-membership predicates. A typical analysis using SMTQuery might look as follows:

1. Collect, for each instance, the answers given by all solvers included in our tool. The supposedly-correct answer for this instance is the one given by the majority of these solvers in `UNSAT` cases and indicated by a correct model in case of `SAT` instances. We pose

   Extract `InstanceTable` **From** `*` **Where** `((isCorrect(CVC5)` **and** `isCorrect(Z3Str3))` **and** `isCorrect(Z3Seq))`

   to SMTQuery. After 17 minutes our terminal displays the table shown in Figure 6. The output might differ depending on the initialisation of the instances and scheduling of the processes.
2. Select all instances where cvc5 gives the right answer and either Z3Str3 returns the wrong answer or it gives the right answer slower:

   Select `Name` **From** `*` **Where** `((isCorrect(CVC5)` **and** `(`**not** `isCorrect(Z3Str3)))` **or** `(isCorrect(Z3Str3)` **and** `isFaster(CVC5,Z3Str3)))`.

   We get a list of all 94676 matching instances within our benchmarks. This process took about 15 minutes.
3. Count how many of the instances computed in step 2 contain regular-membership predicates:

   Extract `Count` **From** `*` **Where** `(((isCorrect(CVC5)` **and** `(`**not** `isCorrect(Z3Str3)))` **or** `(isCorrect(Z3Str3)` **and** `isFaster(CVC5, Z3Str3)))` **and** `hasRegex)`.

   Again, in roughly 15 minutes we get the following answer: "Total matching ↪ instances: 47070 of 114468 within the selected set (41.12%)".
4. Select for Z3Str3 all instances where Z3Str3 gives the right answer and either cvc5 gives the wrong answer or it gives the right answer slower:

   Select `Name` **From** `*` **Where** `((isCorrect(Z3Str3)` **and** `(`**not** `isCorrect(CVC5)))` **or** `(isCorrect(CVC5)` **and** `isFaster(Z3Str3,CVC5)))`.

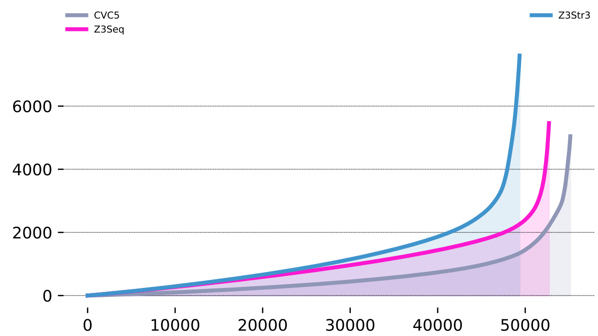


Figure 7: Cactus plot for query `Extract CactusPlot From * Where hasRegex`.

SMTQuery prints a list of 18596 instances within 16 minutes.

5. Count how many of the instances computed in step 4 contain regular-membership predicates:

   Extract `Count` **From** * **Where** `(((isCorrect(Z3Str3)` **and** **(not** `isCorrect(CVC5)))` **or** `(isCorrect(CVC5)` **and** `isFaster(Z3Str3,CVC5)` `))` **and** `hasRegex).`

   In 17 minutes SMTQuery responds with "Total matching instances: 9189 of ↪ 114468 within the selected set (8.02%)."

This analysis gives and substantiates an intuition that cvc5 is more reliable than Z3 and seems to have a better performance when targeting instances that contain regular-membership predicates. SMTQuery offers the export of cactus plots allowing review of the results in a visually appealing way. Figure 7 depicts the cactus plot obtained by posing the query **Extract** `CactusPlot` **From** * **Where** `hasRegex`. We obtained this plot in roughly 3 minutes. The cactus plot shows the cumulative time in seconds taken by each solver on all cases in increasing order of runtime. Solvers that are further to the right and closer to the bottom of the plot have better performance. The plot itself shows that cvc5 seems to implement the most successful algorithm when targeting regular-membership predicates w.r.t. the analysed instances and embedded solvers.

We have presented so far two out of many different possibilities of leveraging information out of a set of benchmarks. Another use case could be the development of a learning algorithm which detects the best solver, from a given set, for each instance by simply extracting features of the instances and establish a correspondence between these features and the fastest solver which produces a correct answer on that instance. Thus, we could obtain decision trees guiding the selection of solvers on certain data, according to some numerical features of the instance, which can be extracted with our tool.

We conclude this section by giving a concrete example where a prototype version of our solver was used in a theoretical investigation (rather unrelated to the area of string solving). In [17], motivated by applications to the processing of event streams [3, 20, 24, 31], the authors study occurrences of subsequences in texts, such that the gaps between the positions of the text matching the symbols of the searched subsequences are subject to both regular and length constraints. To decide how to represent these constraints, the authors of [17] were interested in the regular and length constraints present in benchmarks (appearing alone or in conjunction with other types of constraints). In particular, regarding the regular constraints, the respective investigation was focused on their complexity (length of the regex specifying them, as well the number of states in a minimal deterministic finite automation accepting them). This investigation considered the Kaluza benchmark only, and the following interesting results were reported, whose extraction was done using SMTQuery. The Kaluza benchmark contains 47305 string solving instances, out of which 20740 (i.e., around 43%) contain regular constraints. In total, there are 207038 regular constraints (specified as regular expressions) appearing in these instances. The length of each of these regexes is upper bounded by 20, but the average length of a regex occurring in Kaluza is ≤ 8. Additional processing of the extracted regexes showed that the NFAs cannonically constructed from them have, in average, 17 states, but 99% of the minimal DFAs corresponding to these regexes have at most 20 states, and the average number of states in these minimal DFAs is lower than 11. The authors of [17] also used SMTQuery to investigate how often both regular and length constraints are used in conjunction in the Kaluza benchmark and reported the following results. In this benchmark, 20740 instances contain regular constraints (approximatively 43% of the total number of instances) and 21246 instances contain length constraints (approximatively 44%), and there are 19812 instances which contain both types of constraints; this corresponds to 95% of the instances containing regular constraints, and to 93% of the instances containing length constraints. Based on the information extracted using SMTQuery, the authors of [17] motivate the choice of the model they use to represent regular constraints in their setting (by DFAs rather than regexes) as well as their choice to accommodate both regular and length constraints simultaneously.

5 Conclusion and Future Work

In this paper, we have introduced a benchmark analysis framework called SMTQuery to analyse string constraints and string solvers. Our toolbox provides a query language allowing the exploration of a custom benchmark set. SMTQuery 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 natural directions of development are to also offer built-in coverage for more general theories (e.g., including the closely related theory of bit-vectors), 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. Additionally, our goal is to speed up all sorts of queries, e.g. by smartly combining predicates using the SQLite database or using pre-calculated data more effectively.

References

1. Abdulla, P.A., Atig, M.F., Chen, Y.F., Holík, L., Rezine, A., Rümmer, P., Stenman, J.: Norn: An SMT solver for string constraints. In: International conference on Computer Aided Verification. pp. 462–469. Springer (2015)
2. Amadini, R.: A survey on string constraint solving. ACM Computing Surveys (CSUR) **55**(1), 1–38 (2021)
3. Artikis, A., Margara, A., Ugarte, M., Vansummeren, S., Weidlich, M.: Complex event recognition languages: Tutorial. In: Proc. DEBS 2017. pp. 7–10 (2017). https://doi.org/10.1145/3093742.3095106
4. Aydin, A., Bang, L., Bultan, T.: 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. pp. 255–272. Springer (2015)
5. Barbosa, H., Barrett, C., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., et al.: cvc5: a versatile and industrial-strength smt solver. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 415–442. Springer (2022)
6. 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)
7. Berzish, M., Day, J.D., Ganesh, V., Kulczynski, M., Manea, F., Mora, F., Nowotka, D.: String theories involving regular membership predicates: From practice to theory and back. In: WORDS 2021. pp. 50–64. Springer (2021)
8. Berzish, M., Ganesh, V., Zheng, Y.: Z3str3: A string solver with theory-aware heuristics. In: 2017 Formal Methods in Computer Aided Design (FMCAD). pp. 55–59. IEEE (2017)
9. 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)
10. Beyer, D.: Reliable and reproducible competition results with benchexec and witnesses (report on sv-comp 2016). In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 887–904. Springer (2016)
11. 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)
12. Chen, T., Chen, Y., Hague, M., Lin, A.W., Wu, Z.: What is decidable about string constraints with the replaceall function. Proc. ACM Program. Lang. **2**(POPL), 3:1–3:29 (2018)
13. 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)
14. Chen, T., Hague, M., Lin, A.W., Rümmer, P., Wu, Z.: Decision procedures for path feasibility of string-manipulating programs with complex operations. Proc. ACM Program. Lang. **3**(POPL), 49:1–49:30 (2019)
15. Choffrut, C., Karhumäki, J.: Combinatorics of words. In: Handbook of formal languages, pp. 329–438. Springer (1997)
16. Day, J.D., Ehlers, T., Kulczynski, M., Manea, F., Nowotka, D., Poulsen, D.B.: On solving word equations using SAT. In: International Conference on Reachability Problems. pp. 93–106. Springer (2019)

17. Day, J.D., Kosche, M., Manea, F., Schmid, M.L.: Subsequences with gap constraints: Complexity bounds for matching and analysis problems. CoRR **abs/2206.13896** (2022). https://doi.org/10.48550/ARXIV.2206.13896, to appear in the Proceedings of ISAAC 2022.
18. Day, J.D., Kulczynski, M., Manea, F., Nowotka, D., Poulsen, D.B.: Rule-based word equation solving. In: Proceedings of the 8th International Conference on Formal Methods in Software Engineering. pp. 87–97 (2020)
19. De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: TACAS 2008. pp. 337–340. Springer (2008)
20. Giatrakos, N., Alevizos, E., Artikis, A., Deligiannakis, A., Garofalakis, M.N.: Complex event recognition in the big data era: a survey. VLDB J. **29**(1), 313–352 (2020). https://doi.org/10.1007/s00778-019-00557-w
21. Hague, M.: Strings at MOSCA. ACM SIGLOG News **6**(4), 4–22 (2019)
22. 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
23. Kan, S., Lin, A.W., Rümmer, P., Schrader, M.: Certistr: A certified string solver (technical report). CoRR **abs/2112.06039** (2021, to appear in CPP 2022)
24. Kleest-Meißner, S., Sattler, R., Schmid, M.L., Schweikardt, N., Weidlich, M.: Discovering event queries from traces: Laying foundations for subsequence-queries with wildcards and gap-size constraints. In: Proc. ICDT 2022. LIPIcs, vol. 220, pp. 18:1–18:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022). https://doi.org/10.4230/LIPIcs.ICDT.2022.18
25. 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)
26. Le, Q.L., He, M.: A decision procedure for string logic with quadratic equations, regular expressions and length constraints. In: APLAS. Lecture Notes in Computer Science, vol. 11275, pp. 350–372. Springer (2018)
27. Mora, F., Berzish, M., Kulczynski, M., Nowotka, D., Ganesh, V.: Z3str4: A Multi-armed String Solver. In: FM 2021. pp. 389–406. Springer (2021)
28. Python Software Foundation: pickle — python object serialization. `https://docs.python.org/3/library/pickle.html`, accessed: 2022-01-21
29. The SMT-LIB Initiative: The SMT Library. `https://smtlib.cs.uiowa.edu/benchmarks.shtml`, accessed: 2022-01-17
30. Yu, F., Alkhalaf, M., Bultan, T.: 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. pp. 154–157. TACAS'10, Springer-Verlag, Berlin, Heidelberg (2010)
31. Zhang, H., Diao, Y., Immerman, N.: On complexity and optimization of expensive queries in complex event processing. In: Proc. SIGMOD 2014. pp. 217–228 (2014). https://doi.org/10.1145/2588555.2593671
32. Zheng, Y., Zhang, X., Ganesh, V.: Z3-str: A z3-based string solver for web application analysis. In: ESEC/SIGSOFT FSE 2013. pp. 114–124 (2013)

A Appendix

For clarity, we restate here the link to SMTQuery: `http://smtquery.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 qlang predicates, functions, and extractors

In our implementation of SMTQuery, 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. SMTQuery 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 SMTQuery.

A.2 Using SMTQuery

In this section, we explain the basic commands of SMTQuery and showcase some applications of SMTQuery ranging from simple to more sophisticated experiments. This information is also available on our website.

SMTQuery provides a single executable located at `bin/smtquery` allowing to access all features of our toolbox by positional arguments. We run SMT-Query 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 CVC5 woorpje track01 01_track_1.smt` runs cvc5 on instance `01_track_1.smt` of track `track01` of the `woorpje` benchmark set.

The next paragraphs list some of the currently implemented 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.

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 [Berzish et al.(2021a)Berzish, Day, Ganesh, Kulczynski, Manea, Mora, and Nowotka]).
`isSimpleRegexConcatenation`: filters to all instances which are of the simple regular expression fragment with concatenation (see [Berzish et al.(2021a)Berzish, Day, Ganesh, Kulczynski, Manea, Mora, and Nowotka]).
`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 $s \in \{\textsc{cvc5}, \textsc{Z3Str3}, \textsc{Z3Seq}\}$ declared satisfiable.
- `isUNSAT(`s`)`: filters all instances where $s \in \{\textsc{cvc5}, \textsc{Z3Str3}, \textsc{Z3Seq}\}$ declared unsatisfiable.
- `hasValidModel(s)`: filters all instances where $s \in \{\textsc{cvc5}, \textsc{Z3Str3}, \textsc{Z3Seq}\}$ returned SAT with a valid model.
- `isCorrect(s)`: filters all instances where $s \in \{\textsc{cvc5}, \textsc{Z3Str3}, \textsc{Z3Seq}\}$ returned SAT with a valid model or UNSAT was returned by the majority of used solvers.
- `isFaster(s1,s2)`: filters all instances where $s1, s2 \in \{\textsc{cvc5}, \textsc{Z3Str3}, \textsc{Z3Seq}\}$ 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.
- `ReduceNegations`: shortens sequences of not, keeping the original polarity.
- `EqualsTrue`: simplifies constraints comparing boolean expressions to true.

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.3 Further Examples

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 SMTQuery to visualize the interactions between variables within a formula, using the AST data structure.

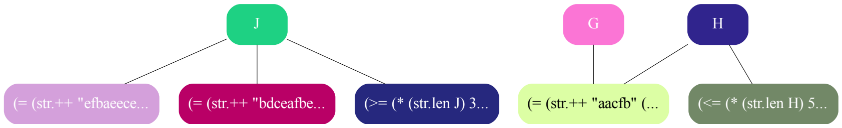


Figure 8: Example variable dependency plot.

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.

Posing the query, e.g., have the instance of Listing 1.2 leading to the plot of Figure 8 where variables G, H and J are top nodes and the corresponding assertions are to bottom nodes.

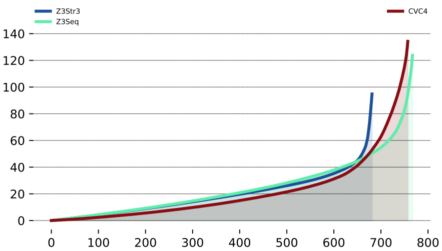
```
(set-logic QF_S)
(declare-fun H () String)
(declare-fun G () String)
(declare-fun J () String)
(assert (= (str.++ "aacfb" G "abdeddaaa") (str.++ "aacfbdffebaaaaac"
    ↪ H "aaa") ))
(assert (= (str.++ "efbaeecedaaecfceffaffaedfcebcf" J "aeaadcbe") (
    ↪ str.++ "e" J "aeecedaaecfceffaffaedfcebcf" 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 the performance of a string solver.* To review the performance of a particular solver, one is usually interested in getting a comparison with respect to other solvers. SMTQuery allows exporting a summary table and the commonly used cactus plot to compare string solvers on benchmarks. For instance, we want to see whether cvc5 is performing well enough on the Woorpje benchmark set. First, we need to trigger cvc5 on the respective benchmark by executing, e.g., **Select** `Name` **From** `woorpje where isSAT(CVC5)`. 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`. The results are visualized in Figure 9.

§ *Modifying Instances.* Analyzing the real-world benchmarks with respect to 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




```
                             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
```

Figure 9: Cactus plot and terminal results for an example query.

can use any extraction function implemented to acquire the data which we are interested in.

§ *Finding and analyzing sub-theories.* In [Berzish et al.(2021a)Berzish, Day, Ganesh, Kulczynski, Manea, Mora, and Nowotka] we have analyzed a large set of benchmarks with respect to 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 [Berzish et al.(2021b)Berzish, Kulczynski, Mora, Manea, Day, Nowotka, and Ganesh]. The inspection of the respective benchmarks was performed using a sequence of different handcrafted scripts, restricted to a particular use case. SMTQuery 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 SMTQuery. 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 [Berzish et al.(2021b)Berzish, Kulczynski, Mora, Manea, Day, Nowotka, and Ganesh], 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.* SMTQuery 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`.

References

Berzish et al.(2021a)Berzish, Day, Ganesh, Kulczynski, Manea, Mora, and Nowotka. 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, 2021a.

Berzish et al.(2021b)Berzish, Kulczynski, Mora, Manea, Day, Nowotka, and Ganesh. 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, 2021b.