An SMT-LIB Format for Sequences and Regular Expressions
(Extended Abstract)

Nikolaj Bjørner¹, Vijay Ganesh², Raphaël Michel³ and Margus Veanes⁴

¹ Microsoft Research
² MIT
³ University of Namur
⁴ Microsoft Research

Abstract

Strings are ubiquitous in software. Tools for verification and testing of software rely in various degrees on reasoning about strings. Web applications are particularly important in this context since they tend to be string-heavy and have large number security errors attributable to improper string sanitization and manipulations. In recent years, many string solvers have been implemented to address the analysis needs of verification, testing and security tools aimed at string-heavy applications. These solvers support a basic representation of strings, functions such as concatenation, extraction, and predicates such as equality and membership in regular expressions. However, the syntax and semantics supported by the current crop of string solvers are mutually incompatible. Hence, there is an acute need for a standardized theory of strings (i.e., SMT-LIBization of a theory of strings) that supports a core set of functions, predicates and string representations.

This paper presents a proposal for exactly such a standardization effort, i.e., an SMT-LIBization of strings and regular expressions. It introduces a theory of sequences generalizing strings, and builds a theory of regular expressions on top of sequences. The proposed logic \( \text{QF}_{\text{BVRE}} \) is designed to capture a common substrate among existing tools for string constraint solving.

1 Introduction

This paper is a design proposal for an SMT-LIB format for a theory of strings and regular expressions. The aim is to develop a set of core operations capturing the needs of verification, analysis, security and testing applications that use string constraints. The standardized theory should be rich enough to support a variety of existing and as-yet-unknown new applications. More complex functions/predicates should be easily definable in it. On the other hand, the theory should be as minimal as possible in order for the corresponding solvers to be relatively easy to write and maintain.

Strings can be viewed as monoids (sequences) where the main operations are creating the empty string, the singleton string and concatenation of strings. Unification algorithms for strings have been subject to extensive theoretical advances over several decades. Modern programming environments support libraries that contain a large set of string operations. Applications arising from programming analysis tools use the additional vocabulary available in libraries. A realistic interchange format should therefore support operations that are encountered in applications.

The current crop of string solvers [9, 12, 3] have incompatible syntax and semantics. Hence, the objective of creating an SMT-LIB format for string and regular expression constraints is to identify a uniform format that can be targeted by applications and consumed by solvers.
The paper is organized as follows. Section 2 introduces the theory Seq of sequences. The theory RegEx of regular expressions in Section 3 is based on Seq. The theories admit sequences and regular expressions over any type of finite alphabet. The characters in the alphabet are defined over the theory of bit-vectors (Section 4). Section 5 surveys the state of string-solving tools. Section 6 describes benchmark sets made available for QF_BVRE and a prototype. We provide a summary in Section 7.

2 Seq: A Theory of Sequences

In the following, we develop Seq as a theory of sequences. It has a sort constructor Seq that takes the sort of the alphabet as argument.

2.1 The Signature of Seq

(par (A) (seq-unit (A) (Seq A))) ; String consisting of a single character
(par (A) (seq-empty (Seq A))) ; The empty string
(par (A) (seq-concat ((Seq A) (Seq A)) (Seq A))) ; String concatenation
(par (A) (seq-cons (A (Seq A)) (Seq A))) ; pre-pend a character to a seq
(par (A) (seq-rev-cons ((Seq A) (Seq A)) (Seq A))) ; post-pend a character
(par (A) (seq-head ((Seq A)) A)) ; retrieve first character
(par (A) (seq-tail ((Seq A)) (Seq A))) ; retrieve tail of seq
(par (A) (seq-last ((Seq A)) A)) ; retrieve last character
(par (A) (seq-first ((Seq A)) (Seq A))) ; retrieve all but the last char
(par (A) (seq-prefix-of ((Seq A) (Seq A)) Bool)) ; test for seq prefix
(par (A) (seq-suffix-of ((Seq A) (Seq A)) Bool)) ; test for postfix
(par (A) (seq-subseq-of ((Seq A) (Seq A)) Bool)) ; sub-sequence test
(par (A) (seq-extract ((Seq A) Num Num) (Seq A))) ; extract sub-sequence
(par (A) (seq-nth ((Seq A) Num) A)) ; extract n’th character
(par (A) (seq-length ((Seq A)) Int)) ; retrieve length of sequence

The sort Num can be either an integer or a bit-vector. The logic QF_BVRE instantiates the sort Num to bit-vectors, and not to an integer.

2.2 Semantics Seq

The constant seq-empty and function seq-concat satisfy the axioms for monoids. That is, seq-empty is an identity of seq-concat and seq-concat is associative.

\[
(seq\text{-}concat seq\text{-}empty x) = (seq\text{-}concat x seq\text{-}empty) = x
\]
\[
(seq\text{-}concat x (seq\text{-}concat y z)) = (seq\text{-}concat (seq\text{-}concat x y) z)
\]

Furthermore, Seq is the theory all of whose models are an expansion to the free monoid generated by seq-unit and seq-empty.
2.2.1 Derived operations

All other functions (except extraction and lengths) are derived. They satisfy the axioms:

\[
\begin{align*}
(\text{seq-cons } x \ y) &= (\text{seq-concat } (\text{seq-unit } x) \ y) \\
(\text{seq-rev-cons } y \ x) &= (\text{seq-concat } y \ (\text{seq-unit } x)) \\
(\text{seq-head } (\text{seq-cons } x \ y)) &= x \\
(\text{seq-tail } (\text{seq-cons } x \ y)) &= y \\
(\text{seq-last } (\text{seq-rev-cons } x \ y)) &= y \\
(\text{seq-first } (\text{seq-rev-cons } x \ y)) &= x \\
(\text{seq-prefix-of } x \ y) &\iff \exists z . (\text{seq-concat } x \ z) = y \\
(\text{seq-suffix-of } x \ y) &\iff \exists z . (\text{seq-concat } z \ x) = y \\
(\text{seq-subseq-of } x \ y) &\iff \exists z,u . (\text{seq-concat } u \ x \ z) = y
\end{align*}
\]

Observe that the value of (\text{seq-head } \text{seq-empty}) is undetermined. Similarly for \text{seq-tail}, \text{seq-first} and \text{seq-last}. Their meaning is under-specified. Thus, the theory \text{Seq} admits all interpretations that satisfy the free monoid properties and the axioms above.

2.2.2 Extraction and lengths

It remains to provide semantics for sequence extraction and length functions. We will here describe these informally.

\textbf{(seq-length } s) \textbf{ The length of sequence } s. \textbf{ Seq satisfies the monoid axioms and is freely generated by unit and concatenation. So every sequence is a finite concatenation of units (i.e., characters in the alphabet). The length of a sequence is the number of units in the concatenation.}

\textbf{(seq-extract } \text{seq } \text{lo } \text{hi) produces the sub-sequence of characters between \text{lo } \text{and } \text{hi-1}. If the length of \text{seq} is less than \text{lo}, then the produced subsequence is empty. If the bit-vector \text{hi} is smaller than \text{lo} the result is, once again, the empty sequence. If the length of \text{seq} is larger than \text{lo}, but less than \text{hi}, then the result is truncated to the length of \text{seq}. In other words, seq-extract satisfies the equation (The length function is abbreviated as } l(s)):

\[
(\text{seq-extract } s \ \text{lo } \text{hi}) =
\begin{cases}
\text{seq-empty} & \text{if } l(s) < \text{lo} \\
\text{seq-empty} & \text{if } \text{hi} < \text{lo} \\
\text{seq-empty} & \text{if } \text{hi} < 0 \\
(\text{seq-extract } (\text{seq-tail } s) \ (\text{lo} - 1) \ (\text{hi} - 1)) & \text{if } 0 < \text{lo} \\
(\text{seq-extract } (\text{seq-first } s) \ (0) \ (m)) & \text{if } 0 < l(s) - \text{hi} + 1 \\
s & \text{otherwise}
\end{cases}
\]

\textbf{(seq-nth } s \ n) \textbf{ Extract the } n^\text{th} \text{ character of sequence } s. \textbf{ Indexing starts at 0, so for example is } c \text{ (where Num ranges over Int).}
3 RegEx: A Theory of Regular Expressions

We summarize a theory of regular expressions over sequences. It includes the usual operations over regular expressions, but also a few operations that we found useful from applications when modeling recognizers of regular expressions. It has a sort constructor \texttt{RegEx} that takes a sort of the alphabet as argument.

3.1 The Signature of \texttt{RegEx}

\[
\begin{align*}
\text{(par } A \text{) (re-empty-set } () \text{) (RegEx } A\text{))} & \quad ; \text{ Empty set} \\
\text{(par } A \text{) (re-full-set } () \text{) (RegEx } A\text{))} & \quad ; \text{ Universal set} \\
\text{(par } A \text{) (re-concat } ((\text{RegEx } A) \text{) (RegEx } A\text{))} & \quad ; \text{ Concatenation} \\
\text{(par } A \text{) (re-of-seq } ((\text{Seq } A) \text{) (RegEx } A\text{))} & \quad ; \text{ Regular expression of sequence} \\
\text{(par } A \text{) (re-empty-seq } () \text{) (RegEx } A\text{))} & \quad ; \text{ same as (re-of-seq-empty)} \\
\text{(par } A \text{) (re-star } ((\text{RegEx } A) \text{) (RegEx } A\text{))} & \quad ; \text{ Kleene star} \\
\text{(par } A \text{) (\text{re-loop } i \text{) } j \text{) (RegEx } A\text{))} & \quad ; \text{ Bounded star, } i,j \geq 0 \\
\text{(par } A \text{) (re-plus } ((\text{RegEx } A) \text{) (RegEx } A\text{))} & \quad ; \text{ Kleene plus} \\
\text{(par } A \text{) (re-option } ((\text{RegEx } A) \text{) (RegEx } A\text{))} & \quad ; \text{ Option regular expression} \\
\text{(par } A \text{) (re-range } (A \text{) A) (RegEx } A\text{))} & \quad ; \text{ Character range} \\
\text{(par } A \text{) (re-union } ((\text{RegEx } A) \text{) (RegEx } A\text{)) (RegEx } A\text{))} & \quad ; \text{ Union} \\
\text{(par } A \text{) (re-difference } ((\text{RegEx } A) \text{) (RegEx } A\text{)) (RegEx } A\text{))} & \quad ; \text{ Difference} \\
\text{(par } A \text{) (re-intersect } ((\text{RegEx } A) \text{) (RegEx } A\text{)) (RegEx } A\text{))} & \quad ; \text{ Intersection} \\
\text{(par } A \text{) (re-complement } ((\text{RegEx } A) \text{) (RegEx } A\text{))} (\text{RegEx } A\text{))} & \quad ; \text{ Complement language} \\
\text{(par } A \text{) (re-of-pred } ((\text{Array } A \text{) Bool}) \text{) (RegEx } A\text{))} & \quad ; \text{ Range of predicate} \\
\text{(par } A \text{) (re-member } ((\text{Seq } A) \text{) (RegEx } A\text{)) Bool}) & \quad ; \text{ Membership test}
\end{align*}
\]

Note the following. The function \texttt{re-range} is defined modulo an ordering over the character sort. The ordering is bound in the logic. For example, in the \texttt{QF_BVRE} logic, the corresponding ordering is unsigned bit-vector comparison \texttt{bvule}. While \texttt{re-range} could be defined using \texttt{re-of-pred}, we include it because it is pervasively used in regular expressions. The function \texttt{re-of-pred} takes an array as argument. The array encodes a predicate. No other features of arrays are used, and the intent is that benchmarks that use \texttt{re-of-pred} also include axioms that define the values of the arrays on all indices. For example we can constrain \texttt{p} using an axiom of the form

\[
\text{(assert (forall ((i (_ BitVec 8))) (iff (select p i) (bvule #0A i))))}
\]

3.2 Semantics of \texttt{RegEx}

Regular expressions denote sets of sequences. Assuming a denotation \([s]\) for sequence expressions, we can define a denotation function of regular expressions:
3.3 Anchors

Most regular expression libraries include anchors. They are usually identified using regular expression constants \(^*\) (match the beginning of the string) and \(\$\) (match the end of a string). We were originally inclined to include operators corresponding these constants. In the end, we opted to not include anchors as part of the core. The reasons were that it is relatively straightforward to convert regular expressions with anchor semantics into regular expressions without anchor semantics. The conversion increases the size of the regular expression at most linearly, but in practice much less. If we were to include anchors, the semantics of regular expression containment would also have to take anchors into account. The denotation of regular expressions would then be context dependent and not as straightforward.

We embed regular expressions with anchor semantics into regular expressions with “regular” semantics using the function complete. It takes three regular expressions as arguments, and it is used to convert the regular expression \(e\) with anchors by calling it with the arguments complete\((e, \top, \top)\). Note that the symbol \(\top\) corresponds to re-full-set, and \(\epsilon\) corresponds to re-empty-set.

\[
\begin{align*}
\text{[re-empty-set]} & = \emptyset \\
\text{[re-full-set]} & = \{ s \mid s \text{ is a sequence} \} \\
\text{[(re-concat } x y]\} & = \{ s \cdot t \mid s \in [x], t \in [y] \} \\
\text{[(re-of-seq } s]\} & = \{ \{s\} \} \\
\text{[re-empty-seq]} & = \{\{\text{seq-empty}\}\} \\
\text{[(re-star } x]\} & = \{x\}^\omega = \bigcup_{i=0}^\omega \{x\}^i \\
\text{[(re-plus } x]\} & = \{x\}^+ = \bigcup_{i=1}^\omega \{x\}^i \\
\text{[(re-option } x]\} & = \{x\} \cup \{\{\text{seq-empty}\}\} \\
\text{[\text{(_re-loop } l u \text{ ) } x]\} & = \bigcup_{i=l}^{\infty} \{x\}^i \\
\text{[(re-union } x y]\} & = \{x\} \cup \{y\} \\
\text{[(re-difference } x y]\} & = \{x\} \setminus \{y\} \\
\text{[(re-intersect } x y]\} & = \{x\} \cap \{y\} \\
\text{[(re-complement } x]\} & = \{\{\text{seq-unit } x\}\} \mid a \leq x \leq z \\
\text{[re-of-pred } p]\} & = \{\{\text{seq-unit } x\}\} \mid p[x] \}
\end{align*}
\]
\[
\begin{align*}
\text{complete}(\text{string}, e_1, e_2) &= e_1 \cdot \text{string} \cdot e_2 \\
\text{complete}(x \cdot y, T, T) &= \text{complete}(x, T, \epsilon) \text{complete}(y, \epsilon, T) \\
\text{complete}(x \cdot y, T, \epsilon) &= \text{complete}(x, T, \epsilon) y \\
\text{complete}(x \cdot y, \epsilon, T) &= x \text{complete}(y, \epsilon, T) \\
\text{complete}(\$; e_1, e_2) &= \epsilon \\
\text{complete}(^*, e_1, e_2) &= \epsilon \\
\text{complete}(x + y, e_1, e_2) &= \text{complete}(x, e_1, e_2) + \text{complete}(y, e_1, e_2)
\end{align*}
\]

We will not define \text{complete} for Kleene star, complement or difference. Such regular expressions are normally considered malformed and are rejected by regular expression tools.

4 The logic QF$_{BVRE}$

The logic QF$_{BVRE}$ uses the theory of sequences and regular expressions. It includes the SMT-LIB theory of bit-vectors as well. Formulas are subject to the following constraints:

- Sequences and regular expressions are instantiated to bit-vectors.
- The sort \text{Num} used for extraction and indexing is a bit-vector.
- \text{re-range} assumes the comparison predicate \text{bvule}.
- Length functions can only occur in comparisons with other lengths or numerals obtained from bit-vectors. So while the range of \text{seq-length} is \text{Int}, it is only used in relative comparisons or in comparisons with a number over a bounded range. In other words, we admit the following comparisons (where \(n\) is an integer constant):

\[
(\{<,\leq,=,\geq,>\} (\text{seq-length} x) (\text{seq-length} y))
\]

To maintain decidability, we also require that if a benchmark contains \((\text{seq-length} x)\) it also contains an assertion of the form \((\text{assert} (< (\text{seq-length} x) n))\).

- The sequence operations \text{seq-prefix-of}, \text{seq-suffix-of} and \text{seq-subseq-of} are excluded.

5 String solvers

String analysis has recently received increased attention, with several automata-based analysis tools. Besides differences in notation, which the current proposal addresses, the tools also differ in expressiveness and succinctness of representation for various fragments of (extended) regular expressions. The tools also use different representations and algorithms for dealing with the underlying automata theoretic operations. A comparison of the basic tradeoffs between
automata representations and the algorithms for product and difference is studied in [11], where
the benchmarks originate from a case study in [19].

The Java String Analyzer (JSA) [7] uses finite automata internally to represent strings with
the dk.brics.automaton library, where automata are directed graphs whose edges represent
contiguous character ranges. Epsilon moves are not preserved in the automata but are elimi-
nated upon insertion. This representation is optimized for matching strings rather than finding
strings.

The Hampi tool [16] uses an eager bitvector encoding from regular expressions to bitvector
logic. The Kudzu/Kaluza framework extends this approach to systems of constraints with
multiple variables and supports concatenation [22]. The original Hampi format does not directly
support regular expression quantifiers “at least m times” and “at most n times”, e.g., a regex
a{1,3} would need to be expanded to alaaalaa. The same limitation is true for the core
constraint language of Kudzu [22] that extends Hampi.

The tool presented in [14] uses lazy search algorithms for solving regular subset constraints,
intersection and determinization. The automaton representation is based on the Boost Graph
Library [23] and uses a range representation of character intervals that is similar to JSA.
The lazy algorithms produce significant performance benefits relative to DPRLE [13] and the
original Rex [27] implementation. DPRLE [13] has a fully verified core specification written in
Gallina [8], and an OCaml implementation that is used for experiments.

Rex [27] uses a symbolic representation of automata where labels are represented by predicates.
Such automata were initially studied in the context of natural language processing [21].
Rex uses symbolic language acceptors, that are first-order encodings of symbolic automata into
the theory of algebraic datatypes. The initial Rex work [27] explores various optimizations of
symbolic automata, such as minimization, that make use of the underlying SMT solver to elimi-
nate inconsistent conditions. Subsequent work [26] explores trade-offs between the language
acceptor based encoding and the use of automata-specific algorithms for language intersec-
tion and language difference. The Symbolic Automata library [25] implements the algebra of
symbolic automata and transducers [24]. Symbolic Automata is the backbone of Rex and Bek.1

Table 1 compares expressivity of the tools with an emphasis on regular expression con-
straints. Columns represent supported features. Kleene stands for the operations re-concat,
re-empty-set, re-empty-seq, re-union, and re-star. Boole stands for re-intersect and
re-complement. Σ refers to supported alphabet theories. In Hampi and Kudzu the Boolean
operations over languages can be encoded through membership constraints and Boolean oper-
ations over formulas. In the Symbolic Automata Toolkit, automata are generic and support all
SMT theories as alphabets.

A typical use of re-range is to succinctly describe a contiguous range of characters, such as
all upper case letters or [A-Z]. Similarly, re-of-pred can be used to define a character class
such as \W (all non-word-letter characters) through a predicate (represented as an array). For

1http://research.microsoft.com/bek/
example, provided that $W$ is defined as follows

$$\forall x(W[x] \iff \neg((\text{'A'} \leq x \leq \text{'Z'}) \lor (\text{'a'} \leq x \leq \text{'z'}) \lor (\text{'0'} \leq x \leq \text{'9'}) \lor x = '_'))$$

then $\text{(re-of-pred } W\text{)}$ is the regex that matches all non-word-letter characters. Finally, \text{re-loop} is a succinct shorthand for bounded loops that is used very frequently in regular expressions.

\text{MONA} \cite{10,17} provides decision procedures for several varieties of monadic second–order logic (M2L) that can be used to express regular expressions over words as well as trees. MONA relies on a highly-optimized multi-terminal BDD-based representation for deterministic automata. Mona is used in the PHP string analysis tool Stranger \cite{29} through a string manipulation library.

Other tools include custom domain-specific string solvers \cite{20,28}. There is also a wide range of application domains that rely on automata based methods: strings constraints with length bounds \cite{30}; automata for arithmetic constraints \cite{6}; automata in explicit state model checking \cite{5}; word equations \cite{1,18}; construction of automata from regular expressions \cite{15}. Moreover, certain string constraints based on common string library functions (not using regular expressions) can be directly encoded using a combination of existing theories provided by an SMT solver.

6 A prototype for \textsc{QF\_BVRE} based on the Symbolic Automata Toolkit

This section describes a prototype implementation for \textsc{QF\_BVRE}. It is based on the Symbolic Automata Toolkit \cite{25} powered by Z3. The description sidesteps the current limitation that all terms $s$ of sort (\texttt{Seq} $\sigma$) are converted to terms of sort (\texttt{List} $\sigma$). While lists in Z3 satisfy all the algebraic properties of sequences, only the operations equivalent to \texttt{seq-empty}, \texttt{seq-cons}, \texttt{seq-head}, and \texttt{seq-tail} are (directly) supported in the theory of lists. This also explains why \texttt{seq-concat} and \texttt{seq-length} (as is also noted in Table 1) are currently not supported in this prototype.

To start with, the benchmark file is parsed by using Z3’s API method \texttt{ParseSmtlib2File} providing a Z3 \texttt{Term} object $\varphi$ that represents the AST of the assertion contained in the file. The assertion $\varphi$ is converted into a formula $\text{Conv}(\varphi)$ where each occurrence of a membership constraint ($\text{re-member } s r$) has been replaced by an atom ($\text{Acc}_r$, $s$), where $\text{Acc}_r$ is a new uninterpreted function symbol called the \textit{symbolic language acceptor} for $r$. The symbol $\text{Acc}_r$ is associated with a set of axioms $\text{Th}(r)$ such that, ($\text{Acc}_r$ $s$) holds modulo $\text{Th}(r)$ iff $s$ is a sequence that matches the regular expression $r$. The converted formula $\text{Conv}(\varphi)$ as well as all the axioms $\text{Th}(r)$ are asserted to Z3 and checked for satisfiability.

The core of the translation is in converting $r$ into a \textit{Symbolic Finite Automaton} \texttt{SFA}(r) and then defining $\text{Th}(r)$ as the theory of \texttt{SFA}(r) \cite{20}. The translation uses closure properties of symbolic automata under the following (effective) Kleene and Boolean operations:

- If $A$ and $B$ are SFAs then there is an SFA $A \cdot B$ such that $L(A \cdot B) = L(A) \cdot L(B)$.
- If $A$ and $B$ are SFAs then there is an SFA $A \cup B$ such that $L(A \cup B) = L(A) \cup L(B)$.
- If $A$ and $B$ are SFAs then there is an SFA $A \cap B$ such that $L(A \cap B) = L(A) \cap L(B)$.
- If $A$ is an SFAs then there is an SFA $A^*$ such that $L(A^*) = L(A)^*$.
If \( A \) is an SFAs then there is an SFA \( \overline{A} \) such that \( L(\overline{A}) = L(\overline{A}) \).

The effectiveness of the above operations does not depend on the theory of the alphabet. In SFAs all transitions are labeled by predicates. In particular, a bit-vector range (re-range \( mn \)) is mapped into an anonymous predicate \( \lambda x. (m \leq x \leq n) \) over bit-vectors and a predicate (re-of-pred \( p \)) is just mapped to \( p \). The overall translation SFA(\( r \)) now follows more-or-less directly by induction of the structure of \( r \). The loop construct (re-loop \( mn r \)) is unfolded by using re-concat and re-union. Several optimizations are possible that have been omitted here.

As a simple example of the above translation, consider the regex

\[
\text{utf16} = ^((\[0-\uD7FF\uD800-\uDFFF]\|[\uD800-\uDBFF][\uDC00-\uDFFF])\*)$
\]

that describes valid UTF16 encoded strings. Using the SMT2 format and assuming the defined sort as (_ BitVec 16) the regex is

\[
(\text{re-star } (\text{re-union } (\text{re-union } (\text{re-range } \#x0000 \#x7FF) (\text{re-range } \#xE000 \#xFFFF)) (\text{re-concat } (\text{re-range } \#xD800 \#xDBFF) (\text{re-range } \#xDC00 \#xDFFF))))
\]

The resulting SFA(\( \text{utf16} \)) can be depicted as follows:

The theory \( Th(\text{utf16}) \) contains the following axioms:

\[
\forall y (Acc_{\text{utf16}}(y) \iff (y = \epsilon \lor \neg y = \epsilon \land (head(y) \leq \#x7FF \lor \#xE000 \leq head(y)) \land Acc_{\text{utf16}}(tail(y))) \lor \neg (y \neq \epsilon \land \#x800 \leq head(y) \leq \#xFF \land Acc_{\text{utf16}}(tail(y))))
\]

Benchmarks in the proposed SMT-LIB format that are handled by the tool are available\(^2\).

### 7 Summary

We proposed an interchange format for sequences and regular expressions. It is based on the features of strings and regular expressions used in current main solvers for regular expressions. There are many possible improvements and extensions to this proposed format. For example, it is tempting to leverage that SMT-LIB already allows string literals. The first objective is to identify a logic that allows to exchange meaningful benchmarks between solvers and enable comparing techniques that are currently being developed for solving sequence and regular expression constraints.

#### 7.1 Contributors

Several people contributed to the discussions about SMTization of strings, including Nikolaj Bjørner, Vijay Ganesh, Tim Hinrichs, Pieter Hooimeijer, Raphaël Michel, Ruzica Piskac, Cesare Tinelli, Margus Veanes, Andrei Voronkov and Ting Zhang. This effort grew out from discussions at Dagstuhl seminar \(^2\) and was followed up at strings-smtization@googlegroups.com

\(^2\)http://research.microsoft.com/~nbjorner/microsoft.automata.smtbenchmarks.zip
References

SMT-LIB Sequences and Regular Expressions
N. Bjørner, V. Ganesh, R. Michel, M. Veanes


