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

Nikolaj Bjørner[1], Vijay Ganesh[2], Raphaël Michel[3] and Margus Veanes[4]

[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 sanitzation 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 `QF_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 concatentation 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.

P. Fontaine, A. Goel (eds.), SMT 2012 (EPiC Series, vol. 20), pp. 77–87

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 concatentation

(par (A) (seq-cons (A (Seq A)) (Seq A)))      ; pre-pend a character to a seq
(par (A) (seq-rev-cons ((Seq A) A) (Seq A)))        ; post-pend a characeter
(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
                                                            parametric in Num
(par (A) (seq-nth ((Seq A) Num) A))                  ; extract n'th character
                                                            parametric in Num
(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.

$$(\texttt{seq-concat seq-empty } x) = (\texttt{seq-concat } x \texttt{ seq-empty}) = x$$
$$(\texttt{seq-concat } x\ (\texttt{seq-concat } y\ z)) = (\texttt{seq-concat}\ (\texttt{seq-concat } x\ y)\ z)$$

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:

$$
\begin{aligned}
&(\texttt{seq-cons}\ x\ y) = (\texttt{seq-concat}\ (\texttt{seq-unit}\ x)\ y)\\
&(\texttt{seq-rev-cons}\ y\ x) = (\texttt{seq-concat}\ y\ (\texttt{seq-unit}\ x))\\
&(\texttt{seq-head}\ (\texttt{seq-cons}\ x\ y)) = x\\
&(\texttt{seq-tail}\ (\texttt{seq-cons}\ x\ y)) = y\\
&(\texttt{seq-last}\ (\texttt{seq-rev-cons}\ x\ y)) = y\\
&(\texttt{seq-first}\ (\texttt{seq-rev-cons}\ x\ y)) = x\\
&(\texttt{seq-prefix-of}\ x\ y) \Leftrightarrow \exists z\ .\ (\texttt{seq-concat}\ x\ z) = y\\
&(\texttt{seq-suffix-of}\ x\ y) \Leftrightarrow \exists z\ .\ (\texttt{seq-concat}\ z\ x) = y\\
&(\texttt{seq-subseq-of}\ x\ y) \Leftrightarrow \exists z, u\ .\ (\texttt{seq-concat}\ u\ x\ z) = y
\end{aligned}
$$

Observe that the value of `(seq-head seq-empty)` is undetermined. Similarly for `seq-tail`, `seq-first` and `seq-last`. Their meaning is *under-specified.* Thus, the theory `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.

`(seq-length s)` The length of sequence `s`. `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.

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

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

`(seq-nth s n)` Extract the `n`'th character of sequence `s`. Indexing starts at 0, so for example is c (where `Num` ranges over `Int`).

(`seq-nth` (`seq-cons` c s) 0)

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 `RegEx` that takes a sort of the alphabet as argument.

3.1 The Signature of RegEx

```
(par (A) (re-empty-set () (RegEx A)))                              ; Empty set
(par (A) (re-full-set () (RegEx A)))                            ; Univeral set
(par (A) (re-concat ((RegEx A) (RegEx A)) (RegEx A)))         ; Concatenation
(par (A) (re-of-seq ((Seq A)) (RegEx A)))    ; Regular expression of sequence
(par (A) (re-empty-seq () (RegEx A)))        ; same as (re-of-seq seq-empty)

(par (A) (re-star   ((RegEx A)) (RegEx A)))                     ; Kleene star
(par (A) ((_ re-loop i j) ((RegEx A)) (RegEx A)))    ; Bounded star, i,j >= 0
(par (A) (re-plus   ((RegEx A)) (RegEx A)))                     ; Kleene plus
(par (A) (re-option ((RegEx A)) (RegEx A)))      ; Option regular expression
(par (A) (re-range (A A) (RegEx A)))                        ; Character range

(par (A) (re-union      ((RegEx A) (RegEx A)) (RegEx A)))             ; Union
(par (A) (re-difference ((RegEx A) (RegEx A)) (RegEx A)))        ; Difference
(par (A) (re-intersect  ((RegEx A) (RegEx A)) (RegEx A)))      ; Intersection
(par (A) (re-complement ((RegEx A)) (RegEx A)))        ; Complement language

(par (A) (re-of-pred ((Array A Bool)) (RegEx A)))       ; Range of predicate

(par (A) (re-member ((Seq A) (RegEx A)) Bool))             ; Membership test
```

Note the following. The function `re-range` is defined modulo an ordering over the character sort. The ordering is bound in the logic. For example, in the `QF_BVRE` logic, the corresponding ordering is unsigned bit-vector comparison `bvule`. While `re-range` could be defined using `re-of-pred`, we include it because it is pervasively used in regular expressions. The function `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 `re-of-pred` also include axioms that define the values of the arrays on all indices. For example we can constrain `p` using an axiom of the form

```
(assert (forall ((i (_ BitVec 8))) (iff (select p i) (bvule #0A i))))
```

3.2 Semantics of RegEx

Regular expressions denote sets of sequences. Assuming a denotation $[\![s]\!]$ for sequence expressions, we can define a denotation function of regular expressions:

$$
\begin{array}{rcl}
[\![\texttt{re-empty-set}]\!] & = & \emptyset \\
[\![\texttt{re-full-set}]\!] & = & \{s \mid s \text{ is a sequence}\} \\
[\![(\texttt{re-concat}\ x\ y)]\!] & = & \{s \cdot t \mid s \in [\![x]\!], t \in [\![y]\!]\} \\
[\![(\texttt{re-of-seq}\ s)]\!] & = & \{[\![s]\!]\} \\
[\![\texttt{re-empty-seq}]\!] & = & \{[\![\texttt{seq-empty}]\!]\} \\
[\![(\texttt{re-star x})]\!] & = & [\![x]\!]^* = \bigcup_{i=0}^{\omega} [\![x]\!]^i \\
[\![(\texttt{re-plus x})]\!] & = & [\![x]\!]^+ = \bigcup_{i=1}^{\omega} [\![x]\!]^i \\
[\![(\texttt{re-option x})]\!] & = & [\![x]\!] \cup \{[\![\texttt{seq-empty}]\!]\} \\
[\![((\_\ \texttt{re-loop}\ l\ u)\ x)]\!] & = & \bigcup_{i=l}^{u} [\![x]\!]^i \\
[\![(\texttt{re-union x y})]\!] & = & [\![x]\!] \cup [\![y]\!] \\
[\![(\texttt{re-difference x y})]\!] & = & [\![x]\!] \setminus [\![y]\!] \\
[\![(\texttt{re-intersect x y})]\!] & = & [\![x]\!] \cap [\![y]\!] \\
[\![(\texttt{re-complement x})]\!] & = & \overline{[\![x]\!]} \\
[\![(\texttt{re-range}\ a\ z)]\!] & = & \{[\![(\texttt{seq-unit}\ x)]\!] \mid a \leq x \leq z\} \\
[\![\texttt{re-of-pred}\ p]\!] & = & \{[\![(\texttt{seq-unit}\ x)]\!] \mid p[x]\} \\
[\![(\texttt{re-member}\ s\ x)]\!] & = & [\![s]\!] \in [\![x]\!]
\end{array}
$$

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 funnction *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 ϵ corresponds to `re-empty-set`.

$$
\begin{array}{rcl}
\mathit{complete}(\mathit{string}, e_1, e_2) & = & e_1 \cdot \mathit{string} \cdot e_2 \\
\mathit{complete}(x \cdot y, \top, \top) & = & \mathit{complete}(x, \top, \epsilon)\ \mathit{complete}(y, \epsilon, \top) \\
\mathit{complete}(x \cdot y, \top, \epsilon) & = & \mathit{complete}(x, \top, \epsilon)\ y \\
\mathit{complete}(x \cdot y, \epsilon, \top) & = & x\ \mathit{complete}(y, \epsilon, \top) \\
\mathit{complete}(\$, e_1, e_2) & = & \epsilon \\
\mathit{complete}(\hat{}, e_1, e_2) & = & \epsilon \\
\mathit{complete}(x + y, e_1, e_2) & = & \mathit{complete}(x, e_1, e_2) + \mathit{complete}(y, e_1, e_2)
\end{array}
$$

We will not define *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 `Num` used for extraction and indexing is a bit-vector.
- `re-range` assumes the comparison predicate `bvule`.
- Length functions can only occur in comparisons with other lengths or numerals obtained from bit-vectors. So while the range of `seq-length` is `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):

  $$(\{\texttt{<}, \texttt{<=}, \texttt{=}, \texttt{>=}, \texttt{>}\}\ (\texttt{seq-length}\ x)\ (\texttt{seq-length}\ y))$$
  $$(\{\texttt{<}, \texttt{<=}, \texttt{=}, \texttt{>=}, \texttt{>}\}\ (\texttt{seq-length}\ x)\ n)$$

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

- The sequence operations `seq-prefix-of`, `seq-suffix-of` and `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 eliminated 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 `a|aa|aaa`. 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 eliminate inconsistent conditions. Subsequent work [26] explores trade-offs between the language acceptor based encoding and the use of automata-specific algorithms for language intersection 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]

	Kleene	*Boole*	`re-range`	`re-of-pred`	`re-loop`	`seq-concat`	`seq-length`	Σ
JSA	✓	✓	✓					BV16
Hampi	✓	✓						BV8
Kudzu/Kaluza	✓	✓				✓	✓	BV8
Symbolic Automata/Rex	✓	✓	✓	✓	✓			ALL

Table 1: Expressivity of string tools.

Table 1 compares expressivity of the tools with an emphasis on regular expression constraints. 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 operations 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

[1] `http://research.microsoft.com/bek/`

example, provided that W is defined as follows

$$\forall x(W[x] \Leftrightarrow \neg((\texttt{'A'} \leq x \leq \texttt{'Z'}) \vee (\texttt{'a'} \leq x \leq \texttt{'z'}) \vee (\texttt{'0'} \leq x \leq \texttt{'9'}) \vee x = \texttt{'\_'}))$$

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

MONA [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 [29] through a string manipulation library.

Other tools include custom domain-specific string solvers [20, 28]. There is also a wide range of application domains that rely on automata based methods: strings constraints with length bounds [30]; automata for arithmetic constraints [6]; automata in explicit state model checking [5]; word equations [1, 18]; construction of automata from regular expressions [15]. Moreover, certain string constraints based on common string library functions [4] (not using regular expressions) can be directly encoded using a combination of existing theories provided by an SMT solver.

6 A prototype for `QF_BVRE` based on the Symbolic Automata Toolkit

This section describes a prototype implementation for `QF_BVRE`. It is based on the Symbolic Automata Toolkit [25] powered by Z3. The description sidesteps the current limitation that all terms s of sort (`Seq` σ) are converted to terms of sort (`List` σ). While lists in Z3 satisfy all the algebraic properties of sequences, only the operations equivalent to `seq-empty`, `seq-cons`, `seq-head`, and `seq-tail` are (directly) supported in the theory of lists. This also explains why `seq-concat` and `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 `ParseSmtlib2File` providing a Z3 `Term` object φ that represents the AST of the assertion contained in the file. The assertion φ is converted into a formula $\mathit{Conv}(\varphi)$ where each occurrence of a membership constraint (`re-member` s r) has been replaced by an atom $(\mathit{Acc}_r\ s)$, where Acc_r is a new uninterpreted function symbol called the *symbolic languge acceptor for* r. The symbol Acc_r is associated with a set of axioms $\mathit{Th}(r)$ such that, $(\mathit{Acc}_r\ s)$ holds modulo $\mathit{Th}(r)$ iff s is a sequence that matches the regular expression r. The converted formula $\mathit{Conv}(\varphi)$ as well as all the axioms $\mathit{Th}(r)$ are asserted to Z3 and checked for satisfiability.

The core of the translation is in converting r into a *Symbolic Finite Automaton* $\mathit{SFA}(r)$ and then defining $\mathit{Th}(r)$ as the theory of $\mathit{SFA}(r)$ [26]. 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 $L(\overline{A}) = \overline{L(A)}$.

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` m n) is mapped into an anonymous predicate $\lambda x.(m \le x \le n)$ over bit-vectors and a predicate (`re-of-pred` p) is just mapped to p. The overall translation $\mathit{SFA}(r)$ now follows more-or-less directly by induction of the structure of r. The loop construct (`re-loop` m n r) is unfolded by using `re-concat` and `re-union`. Several optimizatons are possible that have been omitted here.

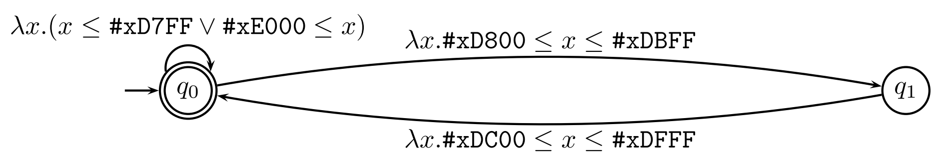
As a simple example of the above translation, consider the regex

$$\mathit{utf16} = \texttt{\^{}([\textbackslash 0-\textbackslash uD7FF\textbackslash uE000-\textbackslash uFFFF]|([\textbackslash uD800-\textbackslash uDBFF][\textbackslash uDC00-\textbackslash uDFFF]))*\$}$$

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

```
(re-star (re-union (re-union (re-range #x0000 #xD7FF) (re-range #xE000 #xFFFF))
                   (re-concat (re-range #xD800 #xDBFF) (re-range #xDC00 #xDFFF))))
```

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



and the theory $\mathit{Th}(\mathit{utf16})$ contains the following axioms:

$$\begin{aligned}
&\forall y(\mathit{Acc}_{\mathit{utf16}}(y) \Leftrightarrow (y = \epsilon \vee \\
&\qquad (y \neq \epsilon \wedge (\mathit{head}(y) \le \texttt{\#xD7FF} \vee \texttt{\#xE000} \le \mathit{head}(y)) \wedge \mathit{Acc}_{\mathit{utf16}}(\mathit{tail}(y))) \vee \\
&\qquad (y \neq \epsilon \wedge \texttt{\#xD800} \le \mathit{head}(y) \le \texttt{\#xDBFF} \wedge \mathit{Acc}_1(\mathit{tail}(y))))) \\
&\forall y(\mathit{Acc}_1(y) \Leftrightarrow (y \neq \epsilon \wedge \texttt{\#xDC00} \le \mathit{head}(y) \le \texttt{\#xDFFF} \wedge \mathit{Acc}_{\mathit{utf16}}(\mathit{tail}(y))))
\end{aligned}$$

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

[1] Sebastian Bala. Regular language matching and other decidable cases of the satisfiability problem for constraints between regular open terms. In *STACS*, pages 596–607, 2004.

[2] Nikolaj Bjørner, Robert Nieuwenhuis, Helmut Veith, and Andrei Voronkov. Decision Procedures in Soft, Hard and Bio-ware - Follow Up (Dagstuhl Seminar 11272). *Dagstuhl Reports*, 1(7):23–35, 2011.

[3] Nikolaj Bjørner, Nikolai Tillmann, and Andrei Voronkov. Path feasibility analysis for string-manipulating programs. In *TACAS*, 2009.

[4] Nikolaj Bjørner, Nikolai Tillmann, and Andrei Voronkov. Path feasibility analysis for string-manipulating programs. In *TACAS*, 2009.

[5] Stefan Blom and Simona Orzan. Distributed state space minimization. *J. Software Tools for Technology Transfer*, 7(3):280–291, 2005.

[6] Bernard Boigelot and Pierre Wolper. Representing arithmetic constraints with finite automata: An overview. In *ICLP 2002: Proceedings of The 18th International Conference on Logic Programming*, pages 1–19, 2002.

[7] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Precise Analysis of String Expressions. In *SAS*, 2003.

[8] Thierry Coquand and Gérard P. Huet. The calculus of constructions. *Information and Computation*, 76(2/3):95–120, 1988.

[9] Vijay Ganesh, Adam Kiezun, Shay Artzi, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. Hampi: A string solver for testing, analysis and vulnerability detection. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *CAV*, volume 6806 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2011.

[10] J.G. Henriksen, J. Jensen, M. Jørgensen, N. Klarlund, B. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In *TACAS'95*, volume 1019 of *LNCS*, 1995.

[11] Pieter Hooimeijer and Margus Veanes. An evaluation of automata algorithms for string analysis. In *VMCAI'11*, volume 6538 of *LNCS*, pages 248–262. Springer, 2011.

[12] Pieter Hooimeijer and Westley Weimer. A decision procedure for subset constraints over regular languages. In *PLDI*, 2009.

[13] Pieter Hooimeijer and Westley Weimer. A decision procedure for subset constraints over regular languages. In *PLDI*, 2009.

[14] Pieter Hooimeijer and Westley Weimer. Solving string constraints lazily. In *ASE*, 2010.

[15] Lucian Ilie and Sheng Yu. Follow automata. *Information and Computation*, 186(1):140–162, 2003.

[16] Adam Kiezun, Vijay Ganesh, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. HAMPI: a solver for string constraints. In *ISSTA*, 2009.

[17] Nils Klarlund, Anders Møller, and Michael I. Schwartzbach. MONA implementation secrets. *International Journal of Foundations of Computer Science*, 13(4):571–586, 2002.

[18] Michal Kunc. What do we know about language equations? In *Developments in Language Theory*, pages 23–27, 2007.

[19] Nuo Li, Tao Xie, Nikolai Tillmann, Peli de Halleux, and Wolfram Schulte. Reggae: Automated test generation for programs using complex regular expressions. In *ASE'09*, 2009.

[20] Yasuhiko Minamide. Static approximation of dynamically generated web pages. In *WWW '05*, pages 432–441, 2005.

[21] Gertjan Van Noord and Dale Gerdemann. Finite state transducers with predicates and identities. *Grammars*, 4:263–286, 2001.

[22] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A Symbolic Execution Framework for JavaScript, Mar 2010.

[23] Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library: User Guide*

and Reference Manual (C++ In-Depth Series). Addison-Wesley Professional, December 2001.

[24] M. Veanes, P. Hooimeijer, B. Livshits, D. Molnar, and N. Bjørner. Symbolic finite state transducers: Algorithms and applications. In *POPL'12*, 2012.

[25] Margus Veanes and Nikolaj Bjørner. Symbolic automata: The toolkit. In C. Flanagan and B. König, editors, *TACAS'12*, volume 7214 of *LNCS*, pages 472–477. Springer, 2012.

[26] Margus Veanes, Nikolaj Bjørner, and Leonardo de Moura. Symbolic automata constraint solving. In C. Fermüller and A. Voronkov, editors, *LPAR-17*, volume 6397 of *LNCS/ARCoSS*, pages 640–654. Springer, 2010.

[27] Margus Veanes, Peli de Halleux, and Nikolai Tillmann. Rex: Symbolic Regular Expression Explorer. In *ICST'10*. IEEE, 2010.

[28] Gary Wassermann and Zhendong Su. Sound and precise analysis of web applications for injection vulnerabilities. In *PLDI*, 2007.

[29] Fang Yu, Muath Alkhalaf, and Tevfik Bultan. Stranger: An automata-based string analysis tool for PHP. In *TACAS'10*, LNCS. Springer, 2010.

[30] Fang Yu, Tevfik Bultan, and Oscar H. Ibarra. Symbolic String Verification: Combining String Analysis and Size Analysis. In *TACAS*, pages 322–336, 2009.