PASS: String Solving with Parameterized Array and Interval Automaton

Guodong Li and Indradeep Ghosh

Fujitsu Labs of America, CA
{gli, ighosh}@fla.fujitsu.com

Abstract. The problem of solving string constraints together with numeric constraints has received increasing interest recently. Existing methods use either bit-vectors or automata (or their combination) to model strings, and reduce string constraints to bit-vector constraints or automaton operations, which are then solved in the respective domain. Unfortunately, they often fail to achieve a good balance between efficiency, accuracy, and comprehensiveness. In this paper we illustrate a new technique that uses parameterized arrays as the main data structure to model strings, and converts string constraints into quantified expressions that are solved through quantifier elimination. We present an efficient and sound quantifier elimination algorithm. In addition, we use an automaton model to handle regular expressions and reason about string values faster. Our method does not need to enumerate string lengths (as bit-vector based methods do), or concrete string values (as automaton based methods do). Hence, it can achieve much better accuracy and efficiency. In particular, it can identify unsatisfiable cases quickly. Our solver (named PASS) supports most of the popular string operations, including string comparisons, string-numeric conversions, and regular expressions. Experimental results demonstrate the advantages of our method.

1 Introduction

A string solver is used to determine the satisfiability of a set of constraints involving string operations. These constraint can be mixed with numeric constraints, in which case we call them *hybrid* constraints. This paper is about how to solve hybrid constraints efficiently using SMT solving and automaton approximation.

Hybrid constraints may be produced by a static analyzer or a symbolic executor. For example, a symbolic executor for web applications may produce thousands of path conditions containing non-trivial hybrid constraints. Solving these constraints efficiently is the key for the tool to be scalable and practical. A typical web application takes string inputs on web pages and performs a lot of string operations such as `concatenation`, `substring`, `>`, and `matches`. There are also three more typical requirements: (1) strings are converted into numeric values for back-end computations; (2) string values are constrained through regular expressions; and (3) unsatisfiable hybrid constraints should be identified quickly. This poses unique challenges to many symbolic execution tools [5, 12, 13, 17] which usually handle only numeric constraints well.

While there are some external string solvers [2, 20, 8, 18, 15, 9] available, none of them meets our need to obtain a good balance between efficiency, accuracy, and comprehensiveness. Roughly, existing solvers can be divided into two categories: (1) bit-vector (BV) based methods, which model a string with a fixed-length bit-vector; and (2) automaton based methods, which model a string with an automaton. A BV method needs to compute the lengths of all strings before constructing bit-vectors, hence it may enumerate all possible length values in order to prove or disprove a set of constraints. Such enumeration often leads to exponential numbers of fruitless trials. In contrast, an automaton method models a string with an automaton capturing all possible values of this string. String automata can be refined according to the relation of the strings. Essentially, an automaton is an *over-approximation* of string values, and the refinement is often insufficient, requiring the enumeration of concrete string values and/or string sequences to find out a valid solution. The methods combining these two models inherit many of the disadvantages while circumvent some.

In this paper we propose a new way to model strings so as to avoid brute-force enumeration of string lengths or values. We model a string with a parameterized array (*parray* for short) such that (1) the array maps indices to character values, (2) both the indices and the characters can be symbolic, and (3) the string length is pure symbolic. With this model, string constraints are converted into quantified constraints (*e.g.* $\exists$ and $\forall$ expressions) which are then handled through our quantifier elimination scheme. Our conversion scheme follows a declarative and non-recursive style. The produced quantified constraints are often beyond the capacity of modern SMT solvers such as Yices [19] and CVC [1]. To handle them, we propose an efficient quantifier elimination algorithm. This conversion scheme is our first contribution. It precisely models string operations and string-numeric conversions. The quantifier eliminator is our second contribution.

Our third contribution is to use interval automata to build an extra model for strings, and reason about string values via automata. We use automata to not only handle regular expressions (RegExps), but also enhance the solving of non-RegExp cases. We demonstrate how to refine the automata through deductive reasoning and fixed-point calculation.

Our fourth contribution is to combine the parray and automaton model to determine satisfiability efficiently. For example, when the automaton domain finds unsat, the solver can safely claim unsat. While the automaton model is mandatory in modeling RegExps, we can use the automata to refine the parray model for locating a solution fast.

We perform preliminary experiments to compare different methods, and show that our method outperforms existing ones in general.

As far as we know, our P-Array based String Solver (PASS) is the first to explicitly use parameterized arrays to model strings and apply quantifier elimination to solve string constraints. It is also the first to combine interval automaton and parray for fast string solving. As for comprehensiveness, it handles virtually all Java string operations, regular expressions, and string-numeric conversions.

```
String s1, s2; int i; // symbolic
if (s1.beginsWith("a1")) {
  if (s2.contains("12")) {
    if ((s1 + s2).endsWith("cd"))
      ...;  // path 1
    else ...;  // path 2
  }
  else if (s2.toLower() > s1)
    { ...; return; } // path 3
  int j = parseInt(s2.substring(i,i+2));
  if (j == 12) ...; // path 4
  else (toString(i) == s2)
    ...; // path 5
  else ...; // path 6
}
else ...; // path 7
      (a)
```

```
String s; // symbolic input
if (s[0] == ‘-’) {
  if (s.match(".\d+,\d{3}"))
    ...;  // path 1
  else {
    int i = s.lastIndexOf(‘,’);
    if (i == -1) ...; // path 2
    else {
      String s1 = s.substring(i+1);
      int x = parseInt(s1);
      if (x > 100 + i)
        ...; // path 3
      else if (s1 < "1000")
        ...; // path 4
      else ...; // path 5
}}}
else ...; // path 6
      (b)
```

Fig. 1. Two example programs producing hybrid constraints.

2 Motivating Examples and Background

Figure 1 shows two Java examples. The first one contains string operations `substring`, `beginsWith`, $>$, *etc.*. Inputs $s1$ and $s2$ are symbolic strings, and i is a symbolic integer. Consider the path conditions (PC) of Path 1 and Path 4. A possible solution for PC1 is $s1$ = "a1" $\wedge$ $s2$ = "12cd". PC4 is unsatisfiable since constraint ¬s2.contains("12") contradicts with the "toInt ..." constraint. Here numeric and strings may be converted back and forth.

$$\begin{array}{ll} \text{PC1}: & \text{s1.beginsWith(``a1'')} \wedge \text{s2.contains(``12'')} \wedge \text{(s1+s2).endsWith(``cd'')} \\ \text{PC4}: & \text{s1.beginsWith(``a1'')} \wedge \neg\text{s2.contains(``12'')} \wedge \text{s2.toLower()} \leq s1 \\ & \wedge\, \text{toInt(s2.substring}(i, i+2)) = 12 \end{array}$$

The second example checks whether the symbolic input s starts with '-'. If yes, it checks whether s is of a popular format depicted by a RegExp (*e.g.* starting with any character, followed by at least one digit, and a comma, and then 3 digits). Then it checks whether ',' appears in s. If yes then the substring after character ',' is taken and converted into an integer x, which is later compared with $100 + i$. This is a typical computation in web applications, *e.g.*, first, performing format checking, then, converting strings to numeric, and finally, branching over the numeric. For example, a valid test case for path 3 is "-,103".

The satisfiability of string+numeric constraints is an undecidable problem in general (see [2] for some discussions). Hence practical solutions are important to tackle string-intensive programs. Existing string solvers cannot fulfill our needs. For example, Microsoft's solver [2] encodes string operations with bit-vector but does not support regular expressions. Hampi [8] and Kaluza [15] also use bit-vector encoding and provide limited support for hybrid constraints and regular expressions. The Rex tool [18] uses automaton and an SMT solver, and represents automaton transitions using logical predicates. Stranger [20] uses an automaton-based method to model string constraints and length bounds for abstract interpretation. A lazy solving technique [11] uses automaton with transitions annotated with integer ranges. A good comparison of the automaton-based approaches is given in [10]. Many of these solvers provide no support or only very limited support for hybrid constraints (*i.e.* combinations of numeric constraints, string constraints, and RegExp constraints). An interested reader may refer to [9] for more discussions. Moreover, even for the supported features, they often use iterations or brute-force enumerations, hence harming the performance. Now we briefly introduce the two main existing string models.

Bit-vector based model [2, 8, 15]. A string of length n is modeled by a bit-vector of $8n$ (or $16n$ for Unicode) bits. Note that n has to be a concrete value before the bit-vector can be instantiated. Hence, a BV method first derives length constraints, then solves them to obtain a concrete assignment to the lengths, and then instantiates the bit-vectors and builds value constraints whose solving gives the final string values. For example, from constraints s1.beginsWith("ab") $\wedge$ s1.contains("12") we can derive $|s1| \geq 2$ (we use notation $||$ for the length), then obtain a concrete length value, *e.g.* $|s1| = 2$, then instantiate a $16b$ bit-vector v and build value constraints $\mathit{extract}(v, 0, 7) =$ 'a' $\wedge$ $\mathit{extract}(v, 8, 15) =$ 'b' $\wedge$ $\mathit{extract}(v, 0, 7) =$ '1' $\wedge$ $\mathit{extract}(v, 8, 15) =$ '2', which is found unsat by the SMT solver. Next, a new length constraint like $|s1| > 2$ is used to start a new iteration. After a few trials a valid solution $s1 =$ "ab12" is found with $|s1| = 4$. Clearly, separating the solving of length constraints and value constraints may result in wasted effort. This is also evidenced by the solving of PC1, where the minimum lengths for $s1$ and $s2$ is 2 and 4 respectively. Since the length constraints specify that $|s1| \geq 2 \wedge |s2| \geq 2 \wedge |s1| + |s2| \geq 2$. there are 2 wasted iterations before the right length values are reached.

The case of the unsatisfiable PC4 is worse. A BV method can infer $|s1| \geq i \ \wedge \ 2 \leq i + 2 \leq |s2|$, then build the value constraints after assigning concrete values to $|s1|$, $|s2|$ and i. After the value constraints are found unsatisfiable, new iterations are performed in an attempt to find a valid solution. Suppose the lengths and i are bounded to 100, then $O(100^3)$ iterations may be needed until time-out occurs. In contrast, our parray method requires no such fruitless iterations and is able to return sat or unsat quickly.

Automaton based model [16, 21, 18, 20, 11, 9]. A string is modeled by an automaton which accepts all possible values of this string. There are two kinds of automata: (1) bit automaton, where each transition is labeled 0 or 1, and a string value is represented by the bits from the start state to an accept state; (2) interval automaton, where each transition represents a character whose value is within an interval (or range) $[lb, ub]$ for lower bound lb and upper bound ub. Since bit automata [21, 20] assembles bit-vectors, here we investigate only interval automata. Note that a bit-automaton method may also require deriving and handling lengths constraints separately from value constraints [21].

Take PC1 for example. Initially, $s1$'s automaton accepts any string starting with "ab"; $s2$'s automaton accepts any string containing "12"; and the automaton concatenating $s1$ and $s2$, say $s3$, accepts any string ending with "cd". We can refine each automaton using the relation between the strings, *e.g.* $s3$'s automaton should also contain "ab" and "12", and $s2$'s automaton should contain "cd". Then $s2$'s shortest solution is "12cd", which is valid.

Unfortunately, although automaton refinement can narrow down the possible values of the strings, it may fail to capture precisely the relation between strings. Consider the following path condition.

$$\text{PC3}: \quad s1 + s2 + s3 = \text{“aaaa”} \wedge s1 \geq s2 \geq s3$$

Obviously, after some (imperfect) refinement we can infer that $s1$'s value can be "", "a", ..., "aaaa". The next step is to assign concrete values to $s1$, $s2$ and $s3$. Suppose we starts with $s1$ = "", then the second constraint enforces $s2 = s3$ = "", which falsifies the first constraint. Similarly $s1$ = "a" does not work. It may take multiple trials before we reach a valid solution like $s1$ = "aa" and $s2 = s3$ = "a". One main problem here is that an automaton represents a set of possible string values, but not the exact relation between strings, *e.g.* only when $s1$ = "" do we know that $s2 = s3$ = "". While such a relation can be encoded in a production of two automata [21], the product-automaton may be too large. Searching strategies and heuristics [11] may help, but are too ad-hoc. We show in this paper a more general and comprehensive technique.

Moreover, the connection between strings modeled by automata and the numeric constraints may be weak. Consider the unsat PC4, where $s2.\text{substring}(i, i+2)$ is converted to an integer for numeric computations. Since both $s2$ and i are symbolic, the values of this expression comprise an infinite set, and encoding them symbolically is not trivial (see Section 4 for more details). As a consequence, an automaton method may find it hard to disprove PC4. In our parray method, no automaton is required to handle PC4, and the unsat result can be obtained without enumerating string lengths or numeric values.

Nevertheless, the automaton model is extremely useful to handle RegExps. We propose a technique to convert automaton representation to parray representation parameterizedly after performing a sophisticated automaton refinement scheme. This scheme is crucial for both the accuracy and the performance.

This work is largely motivated when we built string solvers for Java and JavaScript Web applications. Our automaton-based solver in [9] suffers from above-mentioned issues, which are addressed by PASS.

Overview of our parray based model. A string is modeled by a parray of symbolic length. The main procedure to solve a set of hybrid constraints is:

1. All string constraints not involving regular expressions are converted into equivalent quantified parray constraints (Section 3).
2. If a string is constrained by a regular expression, build a string relation graph for all string variables in the constraints, perform refinement to infer more relations and possible values of the strings (Section 4).

3. For each string associated with a regular expression, build extra parray constraints from this string's automaton (Section 4). If no regular expression is involved in the original hybrid constraints, we can skip steps 2 and 3.
4. Perform quantifier elimination to remove quantifiers iteratively, solve the remaining numeric+array constraints (Section 3.1). This overcomes the limitations of modern SMT solvers like Yices [19].

3 Parameterized Array Based Model

A parameterized array (parray) maps symbolic indices (natural numbers) to symbolic characters. Unquantified parray constraints can be solved by an SMT solver supporting the array theory and the numeric theory. We convert string constraints into quantified parray constraints. Figure 2 shows some simple cases, where the conversions are mostly self-explanatory. Take lastIndexOf for example, integer i marks c's last position in the string. i is either -1 or $< |s|$. If $i = -1$, then $c \notin s$; otherwise, $s[i] = c$. Each character after index i does not equal to c, which is modeled by $\forall n.\, i < n < |s| \Rightarrow s[n] \neq c$.

All exists constraints can be eliminated by introducing fresh variables. Thus all remaining quantified constraints are of format $\forall n.\, n < l \Rightarrow P(n)$ where P is an unquantified constraint or a simple exists constraint. One main rule here is that we avoid using recursions in the conversion. For instance, we may introduce a helper function indexOf' to model indexOf: s.indexOf'$(i, c) = \mathtt{ite}(s[i] = c,\, i,\, s.\text{indexOf'}(i+1, c))$. However this recursive form may bring difficulties in quantifier elimination. Another example is $s > s_1$, whose recursive encoding is easy to specify but hard to solve. We describe below a novel way to encode it.

Figure 3 shows the conversions for some tricky cases, which represent our novel encoding. For i = s.indexOf(s_1), if $i \neq -1$ then i is the first position in s such that s_1 appears, hence s_1 will not appear in any prior position m. The $i = -1$ case is the same as $\neg s$.contains(s_1).

Consider $s_1 = s$.trim(), *i.e.* s_1 is obtained from s by removing all blank characters from the beginning and ending of s. As shown below, we introduce a natural number m to mark the first non-blank character in s. The conversion reads: all characters before m and after $m + |s_1|$ are blank, others are shared by s and s_1 in the same order, with the characters at two ends are not blank.

$s[0]$	...	$s[m-1]$	$s[m]$	...	$s[m+\|s_1\|-1]$	$s[m+\|s_1\|]$	...
' '	...	' '	$s_1[0]$	...	$s_1[\|s_1\|-1]$	' '	...

The conversion of $s > s_1$ is through introducing a natural number m to mark the first position where s and s_1 differs. As shown below, the characters from 0 to $m-1$ are the same. Next, if s_1 is of length m and s's length is greater than s_1's, or $s[m] \neq s_1[m]$, then $s > s_1$. The case of $s \geq s_1$ is similar except that s can equal to s_1, *e.g.* $|s| = |s_1| = m$. The conversions of $s < s_1$ and $s \leq s_1$ are done through $s_1 < s$ and $s_1 \geq s$ respectively.

$s[0]$	$s[1]$	...	$s[m-1]$	$s[m]$
$s_1[0]$	$s_1[1]$	...	$s_1[\|s_1-1\|]$	

Case 1: $m = |s_1|$

$s[0]$	$s[1]$	...	$s[m-1]$	$s[m]$
$s_1[0]$	$s_1[1]$	...	$s_1[m-1]$	$s_1[m]$

Case 2: $|s_1| > m \wedge s[m] > s_1[m]$

String Constraint	**P-Array Constraint**
$s_1 = s_2$	$(\forall n.\, n < \lvert s_1\rvert \Rightarrow s_1[n] = s_2[n]) \wedge \lvert s_1\rvert = \lvert s_2\rvert$
$s_1 \neq s_2$	$(\exists n.\, n < \lvert s_1\rvert \wedge s_1[n] \neq s_2[n]) \vee \lvert s_1\rvert \neq \lvert s_2\rvert$
$s = s_1 + s_2$	$(\forall n.\, n < \lvert s_1\rvert \Rightarrow s_1[n] = s[n]) \wedge$ $(\forall n.\, n < \lvert s_2\rvert \Rightarrow s_2[n] = s[\lvert s_1\rvert + n]) \wedge \lvert s\rvert = \lvert s_1\rvert + \lvert s_2\rvert$
$s_1 = s.\text{substring}(n_1, n_2)$	$(\forall n.\, n < \lvert s_1\rvert \Rightarrow s_1[n] = s[n_1 + n]) \wedge$ $n_1 < n_2 \leq \lvert s\rvert \wedge \lvert s_1\rvert = n_2 - n_1$
$i = s.\text{lastIndexOf}(c)$	$(i = -1 \vee (0 \leq i < \lvert s\rvert \wedge s[i] = c)) \wedge$ $(\forall n.\, i < n < \lvert s\rvert \Rightarrow s[n] \neq c)$
$i = s.\text{indexOf}(c)$	$(i = -1 \vee (0 \leq i < \lvert s\rvert \wedge s[i] = c)) \wedge (\forall n.\, n < i \Rightarrow s[n] \neq c)$
$s.\text{beginsWith}(s_1)$	$(\forall n.\, n < \lvert s_1\rvert \Rightarrow s_1[n] = s[n]) \wedge \lvert s\rvert \geq \lvert s_1\rvert$
$\neg s.\text{beginsWith}(s_1)$	$(\exists n.\, n < \lvert s_1\rvert \wedge s_1[n] \neq s[n]) \vee \lvert s\rvert < \lvert s_1\rvert$
$s.\text{endsWith}(s_1)$	$(\forall n.\, n < \lvert s_1\rvert \Rightarrow s_1[n] = s[\lvert s\rvert - \lvert s_1\rvert + n]) \wedge \lvert s\rvert \geq \lvert s_1\rvert$
$s.\text{contains}(s_1)$	$(\exists m.\, m \leq \lvert s\rvert - \lvert s_1\rvert \wedge (\forall n.\, n < \lvert s_1\rvert \Rightarrow s_1[n] = s[m+n])) \wedge$ $\lvert s\rvert \geq \lvert s_1\rvert$
$\neg s.\text{contains}(s_1)$	$(\forall m.\, m \leq \lvert s\rvert - \lvert s_1\rvert \Rightarrow (\exists n.\, n < \lvert s_1\rvert \wedge s_1[n] \neq s[m+n])) \vee$ $\lvert s\rvert < \lvert s_1\rvert$
$s_1 = s.\text{toUpperCase}()$	$(\forall n.\, n < \lvert s\rvert \Rightarrow s_1[n] = \texttt{ite}(`a' \leq s[n] \leq `z',\, s[n] + `a' - `A',\, s[n]))$ $\wedge \lvert s_1\rvert = \lvert s\rvert$

Fig. 2. Conversion of simple cases (excerpt). Operator $|s|$ denotes string s's length. m and n are natural numbers; i is an integer; c is a character.

String Constraint	**P-Array Constraint**
$i = s.\text{indexOf}(s_1)$ $i \neq -1$	$i \geq 0 \wedge i + \lvert s_1\rvert \leq \lvert s\rvert \wedge (\forall n.\, n < \lvert s_1\rvert \Rightarrow s_1[n] = s[i+n]) \wedge$ $(\forall n.\, n < i \Rightarrow (\exists m.\, m < \lvert s_1\rvert \Rightarrow s_1[n] \neq s[m+n]))$
$s_1 = s.\text{trim}()$	$\exists m.\, m + \lvert s_1\rvert \leq \lvert s\rvert \wedge$ $(\forall n.\, (n < m \vee m + \lvert s_1\rvert \leq n \leq \lvert s\rvert) \Rightarrow s[n] = `\ ') \wedge$ $(\forall n.\, n < \lvert s_1\rvert \Rightarrow s[m+n] = s_1[n]) \wedge$ $s[m] \neq `\ ' \wedge s[m + \lvert s_1\rvert - 1] \neq `\ '$
$s > s_1$	$\exists m.\, m \leq \lvert s_1\rvert \wedge (\forall n.\, n < m \Rightarrow s_1[n] = s[n]) \wedge$ $\lvert s\rvert > m \wedge (\lvert s_1\rvert = m \vee \lvert s_1\rvert > m \wedge s[m] > s_1[m])$
$s \geq s_1$	$\exists m.\, m \leq \lvert s_1\rvert \wedge (\forall n.\, n < m \Rightarrow s_1[n] = s[n]) \wedge$ $(\lvert s\rvert = \lvert s_1\rvert = m \vee \lvert s_1\rvert = m \vee \lvert s_1\rvert > m \wedge s[m] > s_1[m])$
$i = \text{parseInt}(s) \wedge$ $i \geq 0$	$(\lvert s\rvert = 1 \Rightarrow i = s[0] - `0') \wedge$ $(\lvert s\rvert = 2 \Rightarrow i = (s[0] - `0') \times 10 + s[1] - `0') \wedge \ldots \wedge$ $(\lvert s\rvert = 10 \Rightarrow$ $i = ((s[0] - `0') \times 10 + (s[1] - `0')) \times 10 \cdots + (s[9] - `0'))$

Fig. 3. Conversion of more tricky cases (excerpt).

The conversion of parseInt is one of the rare examples where the string length has to be bounded concretely. Since a 32-bit integer can have up to 10 digits, the conversion case splits over the possible length values to produce unquantified constraints. The conversion of parseFloat is similar. In Section 4 we show the automaton model can help infer possible lengths so as to simplify the encoding.

3.1 Solving P-Array Constraints with Quantifier Elimination

The generated forall constraints conform to a specific form: $\forall n.\, L(n) \Rightarrow P(s)$ or $\forall n.\, L(n) \Rightarrow \exists m.\, P(n)$, where P is a non-self-recursive predicate comparing two corresponding elements in two parrays, and L constrains n with respect to string lengths, *e.g.* $n < |s|$. In some cases, this simple format can be handled by a modern SMT solver like Yices [19]. But this is not the case in general. For instance, the most recent Yices version v1.0.36 cannot solve (in 10 minutes) the following simple forall constraints produced from string constraint $s + s =$ "aa", while it can solve $s_1 + s_2 =$ "aa" $\wedge\ |s_1| = |s_2| = 1$.

$$
\begin{aligned}
&2 = |s| + |s| \ \wedge\ \_s_0[0] = \text{`a'} \ \wedge\ \_s_0[1] = \text{`a'} \ \wedge \\
&(\forall n.\, n < |s| \Rightarrow \_s_0[n] = s[n]) \ \wedge\ (\forall n.\, n < |s| \Rightarrow \_s_0[|s| + n] = s[n]) \ .
\end{aligned}
$$

Inspired by the work in [4], we propose an iterative quantifier elimination (QElim) [1] algorithm for the generated constraints. Note that [4] cannot handle most of our parray constraints, *e.g.* when an access's index is $m + n$ or $|s_1| + n$.

The basic idea is to calculate an *index set* and use its elements to instantiate forall constraints so as to eliminate the quantifiers. Given a set of constraints $\mathbb{C}$, parray s's index set (IS) includes all the indices of the accesses to s not bounded by a quantifier. By definition, $\{e \mid s[e] \in \mathbb{C} \wedge \texttt{qnt\_vars}(e) \neq \phi\} \subseteq \text{IS}(s)$, where `qnt_vars` gives the set of quantified variables. That is, for access $s[e]$, if e does not involve any quantifier, then $e \in \text{IS}(s)$. In addition, for each constraint of format $\forall n.\, n < k \Rightarrow P(s[n])$, the upper bound $k - 1$ is in $\texttt{IS}(k)$. This is for taking into the upper bound case into account.

```
Data: Quantified Constraints $\mathbb{C}_q$ + Unquantified Constraints $\mathbb{C}_{uq}$
Result: Unquantified Constraints $\mathbb{C}_{uq}$
forall $s$ do $\texttt{IS}_{old}(s) = \{\}$; calculate $\texttt{IS}(s)$ end
while $\exists s.\, \texttt{IS}_{old}(s) \neq \texttt{IS}(s)$ do
    forall $e \in \texttt{IS}(s) \setminus \texttt{IS}_{old}(s)$ do
        forall $(\forall n.\, L(n) \Rightarrow P(s[f(n)])) \in \mathbb{C}_q$ do
            if sat$(\mathbb{C}_{uq} \wedge L(f^{-1}(e)))$
              add $L(f^{-1}(e)) \Rightarrow P(s[e])$ into $\mathbb{C}_{uq}$
        end
    end
    forall $s$ do $\texttt{IS}_{old}(s) = \texttt{IS}(s)$; append new indices into $\texttt{IS}(s)$ end
end
```

Algorithm 1: Basic QElim algorithm for P-Array Constraints

After s's index set is calculated, we use each element e in it to instantiate constraint $\forall n.\, L(n) \Rightarrow P(s[f(n)])$ by replacing n with $f^{-1}(e)$, where f^{-1} is the inverse function of f. If P is an exists constraint, then its quantifier is removed by introducing a fresh variable. The intuition behind this is: if e matches $f(n)$, then

[1] Strictly speaking, our algorithm is not a conventional QElim which converts quantified constraints to equivalent unquantified ones. Here we reuse this term to indicate that our approach removes or instantiates quantifiers to find solutions.

we need to instantiate n with $f^{-1}(e)$, *i.e.* we should consider the special case $f^{-1}(e)$ for the forall constraint. Note that f is a linear function whose inverse is trivial to compute.

The next steps are described in Algorithm 1. For each new index e in $\texttt{IS}(s)$ but not in $\texttt{IS}_{old}(s)$, we use e to instantiate all forall constraints containing s. For such a constraint, if its assumption is satisfiable upon the current unquantified constraints $\mathbb{C}_{uq}$, then this constraint is added into $\mathbb{C}_{uq}$; otherwise it is ignored. The algorithm continues until no more new indices are found, in which case we remove all forall constraints, leaving a set of quantifier-free constraints.

Consider the above example, in the first round, $\_s_0$'s index set is $\{0, 1, |s|-1\}$. After instantiating the two forall constraints with this set, we obtain six new constraints as following (note that $|s| - 1 < |s|$ is always true).

$$
\begin{aligned}
&(0 < |s| \Rightarrow \_s_0[0] = s[0]) \wedge (1 < |s| \Rightarrow \_s_0[1] = s[1]) \wedge \\
&(0 < |s| \Rightarrow \_s_0[|s|] = s[0]) \wedge (1 < |s| \Rightarrow \_s_0[|s| + 1] = s[1]) \wedge \\
&(|s| - 1 < |s| \Rightarrow \_s_0[|s| - 1] = s[|s| - 1]) \wedge (|s| - 1 < |s| \Rightarrow \_s_0[|s| + |s| - 1] = s[|s| - 1])
\end{aligned}
$$

Since constraint $2 = |s| + |s|$ conflicts with $1 < |s|$, we remove the two new constraints with assumption $1 < |s|$. In the next round, the remaining new constraints give us updated index sets $\texttt{IS}(\_s_0) = \{0, 1, |s| - 1\} \cup \{|s|\}$ and $\texttt{IS}(s) = \{0, |s| - 1\}$. The next index used for instantiation is $|s|$. Since $|s| < |s|$ is false, no new constraints will be added. Now there exists no new index, hence the algorithm terminates. The two forall constraints are removed, resulting in the following final constraints, whose valid solution is $s =$ "a" (and $\_s_0 =$ "aa").

$$
\begin{aligned}
&2 = |s| + |s| \ \wedge \ \_s_0[0] = \text{‘a’} \ \wedge \ \_s_0[1] = \text{‘a’} \wedge \\
&(0 < |s| \Rightarrow \_s_0[0] = s[0]) \ \wedge \ (0 < |s| \Rightarrow \_s_0[|s|] = s[0]) \wedge \\
&(\_s_0[|s| - 1] = s[|s| - 1]) \wedge (\_s_0[|s| + |s| - 1] = s[|s| - 1])
\end{aligned}
$$

The reduction of PC 1 results in the following constraints, which can produce a valid solution $n_1 = 0 \wedge |\_s_0| = 6 \wedge s_1 =$ "a1" $\wedge\ s_2 =$ "12cd".

$$
\begin{aligned}
&|s_1| \geq 2 \wedge |s_2| \geq n_1 + 2 \wedge |\_s_0| = |s_1| + |s_2| \wedge \\
&\_s_0[0] = s_1[0] = \text{‘a’} \wedge \_s_0[1] = s_1[1] = \text{‘1’} \ \wedge \ \_s_0[|\_s_0| - 2] = \text{‘c’} \wedge \_s_0[|\_s_0| - 1] = \text{‘d’} \wedge \\
&\_s_0[|s_1| + n_1] = s_2[n_1] = \text{‘1’} \wedge \_s_0[|s_1| + n_1 + 1] = s_2[n_1 + 1] = \text{‘2’} \wedge \\
&(\_s_0[|s_1| - 1] = s_1[|s_1| - 1]) \wedge (\_s_0[|s_1| + |s_2| - 1] = s_2[|s_2| - 1])
\end{aligned}
$$

For a bounded string (*i.e.* whose length is bounded), its largest index set can contain all indices up to the bound. Hence the algorithm, similar to BV methods, always terminates (a careful reader can realize that our algorithm may terminate faster due to its symbolic index calculation). The obtained $\mathbb{C}_{uq}$ are equiv-satisfiable to the original (quantified + un-quantified) ones. More discussions on the soundness and termination of Algorithm 1 are given in the Appendix. Note that the soundness proof technique in [4] does not apply here since we (1) allow arithmetic operations in array accesses, (2) calculate index-set iteratively, and (3) permit array relations other than =.

Theorem. Algorithm 1 terminates on bounded strings, and generates un-quantified constraints equiv-satisfiable to the original (quantified + un-quantified) ones.

An Optimized Version: Iterative Quantifier Elimination and Solving. Algorithm 1 may unnecessarily compute too many index sets and terminate slowly. Hence in practice we use a slightly revised version shown in Algorithm 2 that can prove sat or unsat much faster. The revisions are on the main procedure of each iteration: (1) we record the new un-quantified constraints in $\mathbb{C}_{cur}$ and use it to compute the new index sets $\texttt{IS}_{cur}$, and then use $\texttt{IS}_{cur}$ to instantiate forall constraints; (2) after the instantiation, solve all un-quantified constraints using an SMT solver, if unsat then the algorithm terminates and safely reports unsat; (3) otherwise we check whether the current solution $solution(\mathbb{C}_{uq})$ satisfies the original string constraints $\mathbb{S}$. If yes then the algorithm terminates with a true valid solution; otherwise go to the next iteration. If the bound *limit* is reached, then return "unknown". Note that the algorithm does not need to iterate over string lengths. In practice only a couple of iterations are needed in most cases.

The soundness of this algorithm is straight-forward, *e.g.* the sat case is warranted by the check on $\mathbb{S}$. We give more details in the Appendix.

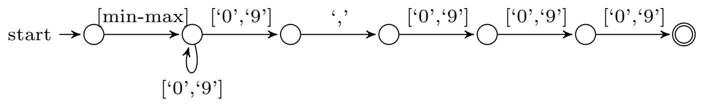
Theorem. Algorithm 2 terminates on bounded strings, and reports sat (or unsat) when the original constraints are indeed sat (or unsat).

```
Data: $\mathbb{C}_q + \mathbb{C}_{uq}$ + String Constraints $\mathbb{S}$
Result: sat, unsat, or unknown
$\texttt{IS}_{cur} = \mathbb{C}_{uq}$ ;
for $i = 0$; $i < limit$; $i{+}{+}$ do
    calculate $\texttt{IS}_{cur}$ w.r.t $\mathbb{C}_{cur}$ ;
    $\mathbb{C}_{cur} = \{\}$ ;
    forall $s$ forall $e \in \texttt{IS}_{cur}(s)$ do
        forall $(\forall n. L(n) \Rightarrow P(s[f(n)])) \in \mathbb{C}_q$
            add $L(f^{-1}(e)) \Rightarrow P(s[e])$ into $\mathbb{C}_{cur}$
    end
    $\mathbb{C}_{uq} = \mathbb{C}_{uq} \cup \mathbb{C}_{cur}$ ;
    if unsat($\mathbb{C}_{uq}$) return unsat ;
    if solution($\mathbb{C}_{uq}$) $\Rightarrow$ sat($\mathbb{S}$) return sat ;
end
return unknown ;
```

Algorithm 2: Iterative QElim

4 Enhancement with Automaton Based Model

We use an interval automaton to represent a string such that all possible values of this string constitutes the language accepted by this automaton. Our implementation is based on the automaton package `dk.brics.automaton` [6]. A transition is labeled the lower bound and upper bound of the associated character. For example, the automaton for the regular expression ".\d+,\d{3}" in the motivating example is shown below.



The implementation of many operations is intuitive. For example, the concatenation of s_1 and s_2 is implemented by adding ϵ transitions from all accepting states of s_1's automaton to all initial states of s_2's automaton, and then removing ϵ to make the resulting automaton deterministic. Many operations such as `intersection` and `minus` are supported by the `dk.brics.automaton` package.

However, we have to model more string operations such as `trim`, `substring`, and `toUpperCase`. For example, `substring(2,4)` returns a substring from index 2 to index 4 (exclusive). To implement it, we first advance 2 transitions from the start state q_0, then mark the reached states as the new start state q_0', and then identify all states reachable from q_0' in 2 transitions as new accepting states. Finally, we intersect this automaton with the one accepting all words of length 2 to get the final automaton. Due to space constraint we will skip the details of implementing the new operations, which we extend from [16, 9].

Automaton Refinement Given a set of string constraints, we build a relation graph with the string automata as the nodes and string relations as the edges. Then we perform iterative refinement to (1) refine each automaton so as to narrow the possible values of the associated string, and (2) derive extra relations.

The first set of refinement rules, including the following, refine automaton values. For better readability, we reuse string names for the automata. Here notation s' denotes the new automaton for s. Operators $\cap$ and $\cdot$ denote intersection and concatenation respectively. Automaton $\mathtt{s_{any}}$ accepts all strings, and $\mathtt{c_{any}}$ accepts any character. We implement some helper operations: $\mathtt{mk\_all\_accept}(s)$ marks all the states in s as accepting states, and $\mathtt{mk\_all\_start}(s)$ marks them as start states. Operation $\mathtt{first}(s, s_2)$ returns the automaton that accepts any string whose concatenation with any string in s_2 is accepted by s. It is implemented over the production of s and s_2 with time complexity O(n^2) for n nodes.

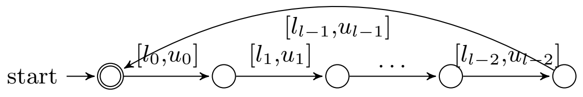
`relation` $\Longrightarrow$	`refinement`
$s = s_1$	$s' = s \cap s_1 \ \wedge \ s_1' = s_1 \cap s$
$s = s_1 + s_2$	$s' = s \cap (s_1 \cdot s_2) \ \wedge \ s_1' = \mathtt{first}(s, s_2) \ \wedge \ s_2' = \mathtt{second}(s, s_1)$
s.beginsWith(s_1)	$s' = s \cap (s_1 \cdot \mathtt{s_{any}}) \ \wedge \ s_1' = s_1 \cap (\mathtt{mk\_all\_accept}(s))$
s.endsWith(s_1)	$s' = s \cap (\mathtt{s_{any}} \cdot s_1) \ \wedge \ s_1' = s_1 \cap (\mathtt{mk\_all\_start}(s))$
s.contains(s_1)	$s' = s \cap (\mathtt{s_{any}} \cdot s_1 \cdot \mathtt{s_{any}}) \wedge$ $s_1' = s_1 \cap (\mathtt{mk\_all\_accept}(\mathtt{mk\_all\_start}(s)))$
$\lvert s \rvert = n$	$s' = s \cap (\mathtt{c_{any}}_0 \cdot \ldots \cdot \mathtt{c_{any}}_{n-1})$

The refinements for `substring`, `lastIndexOf` and `indexOf` are similar to that for `contains`. Some rules are effective with assumptions, *e.g.* length constraint $|s| = n$ or $|s| < n$ is performed only when n is constant. Similarly, we refine some $\neg$ cases only when one of the strings are known to be constant, *e.g.* $\neg s$.contains("abc") $\Longrightarrow$ $s' = s - \{$"abc"$\}$. The refinement process is fixed-point calculation and will stop when no automaton can be refined further.

The second rule set is to refine the relations by inferring new facts from a pair of relations. In general, we may apply source-to-source transformations [14] to simplify the path conditions and then derive new facts. We present below an excerpt of these inference rules, which are repeatedly applied until no more new relation is inferred. They are particularly useful in finding some unsat cases early, *e.g.* $s_1 > s_2 \wedge s_2 \geq s_3 \wedge s_3 \geq s_1$ is unsat. We also define a consumption relation to simplify the relations, *e.g.* with s.beginsWith(s_1) we can safely remove s.contains(s_1).

`relations`	$\Longrightarrow$	`inferred relation`
`op` $\in \{\geq, >\}$: $s_1 > s_2 \wedge s_2$ `op` s_3		$s_1 > s_3$
`op` $\in \{\leq, <, =\}$: $s_1 > s_2 \wedge s_1$ `op` s_2		`false`
s.beginsWith(s_1)		$s \geq s_1$
$\neg s$.contains(s_1) $\wedge$ s.endsWith(s_1)		`false`

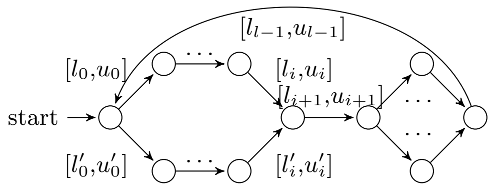
From Automaton to P-Array For constraint s.`matches(`re`)`, s's automaton is refined by regular expression re. When we encode this automaton in the parray domain, the main challenge is on loops. For example, consider the following automaton corresponding to RegExp "([l_0-u_0][l_1-u_1]... [l_{n-1}-u_{n-1}])*", which accepts an infinite set of strings. Clearly, it is impossible to enumerate all the possibilities.

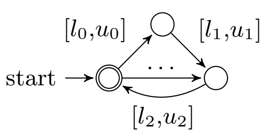


Here we propose a conversion which again uses forall constraints to encode the loop: we introduce a new number m to specify the limit of loop iterations. For each iteration, *e.g.* the n^{th} one, we specify the value interval of each character, *e.g.* the k^{th} character is within $[l_k, u_k]$. Here $n \times l + k$ gives the position of this character in s (l has to be a constant since we use the linear arithmetic of Yices).

$$\exists m. \forall n. \, n < m \Rightarrow \bigwedge_{k=0}^{l-1} (l_k \leq s[n \times l + k] \leq u_k)$$

This encoding method can be generalized to handle *well-formed* loops. A loop is well-formed if it contains no embedded loops and all its sub-sequences between a fork node and the next join are of the same length. A well formed loop is shown below on the left.





Program	set 1		set 2		set 3		set 4		set 5		total	
	pc	T.	pc	T.	pc	T.	pc	T.	pc	T.	#pc	T.
`easychair`	2,3,4	< 0.01s	5	0.03s	1,6,7	0.04s	8	0.06s	9	0.2s	9	0.4s
+our QElim	2,3,4	< 0.01s	1	0.07s	5,6	0.015s	7	0.02s	8,9	0.015s	9	0.2s
`easychair*`	1	0.04	2,5	0.003s	3,6,8	0.004s	4,7	0.005s	9	0.02s	9	0.1s
+our QElim	2,3,4	< 0.01s	1	0.05s	5,6	< 0.01s	7,8	0.01s	9	0.03s	9	0.12s
`lastIndexOf`	1	0.05s	2	0.01s	3,4	0.02s	5	0.1s	6	0.24s	6	0.5s
+our QElim	1,2	0.02s	3	0.015s	4	0.04s	5	0.11s	6	0.25	6	0.46s
+automaton	1	0.03s	2	0.2s	3	0.5s	4	0.35s	5,6	0.65s	6	2.34s
`example (a)`	1	1.02s	2	0.3s	3,7	0.01s	4	0.4s	5,6	0.8s,3.1s	7	7.3s
+our QElim	1	0.04s	2,3	< 0.01s	4,6	0.01s	5	0.1s	7	< 0.01s	7	0.16s
`example (b)`	1	0.23s	2	< 0.01s	3	0.02s	4,6	0.015s	5	0.15s	6	0.45s
+our QElim	1	0.23s	2	< 0.01s	3,4	0.015s	5	0.02s	6	0.01s	6	0.29s

Table 1. Experimental results on toy examples. Here `pc` marks the path condition numbers. Time (T) is measured in seconds.

This loop can be encoded with the following parray constraint.

$$\begin{aligned}&\exists m.\, \forall n.\, n < m \Rightarrow\\ &(\bigwedge_{k=0}^{i}(l_k \leq s[n \times l + k] \leq u_k) \ \vee\ \bigwedge_{k=0}^{i}(l'_k \leq s[n \times l + k] \leq u'_k)) \wedge\\ &(l_{i+1} \leq s[n \times l + i + 1] \leq u_{i+1}) \wedge \ldots\end{aligned}$$

The loop on the right is not well-formed: path 1 $[l_0, u_0] \rightarrow [l_1, u_1]$ and path 2 $[l_2, u_2]$ may alternate in the iterations, *e.g.* `path 1` $\rightarrow$ `path 1` $\rightarrow$ `...` or `path 2` $\rightarrow$ `path 1` $\rightarrow$ `....` For a non well-formed loop, we unroll the iterations to a pre-defined limit and disjoint the paths to produce parray constraints.

Our encoding of a well-formed automaton is complete and sound; but it is incomplete for non well-formed automata. Hence it is crucial to use refinements described in Section 4 to refine the automaton. In many cases non well-formed loops (*e.g.* embedded loops) are refined to well-formed ones. Even if a refined loop is not well-formed, it contains more information for better unrolled paths.

Note that we only convert an automata related to a RegExp All others need not to be converted since they have been modeled precisely in the parray domain. For example, in the motivating example (b), s but not $s1$ will be converted. In example (a), no automaton is needed to built at all (unless we want to use the automaton domain to help the solving)!

The facts obtained from automata can be fed to the parray domain, *e.g.* the minimal string lengths, the known values of the characters at some positions, *etc.*. This can sometimes help the parray encoding and solving.

5 Evaluation Results

Our solver is written in Java. We run it on benchmark programs on a laptop with a 2.40GHz Intel Core(TM)2 Duo processor and 4GB memory. We evaluate the parray solution and the one with parray+automaton.

We first test the main benchmark programs in [2], `easychair` and `lastIndexOf`, as well as the two motivating examples in Section 2. Table 1 shows the results on these toy programs. This includes the results for the comparison with Microsoft's BV-based solver. For `easychair`, with the parray encoding and Yices we can solve all 9 valid paths in 0.4 second ([2] takes about 1 second [2], but we do not include the exception paths that are trivial to compute). With our QElim method (Algorithm 2) the time is reduced to 0.2 second. Since a BV method needs only one iteration to handle these paths, we mutate them by introducing tricky unsat cases, which a BV method may need many iterations to disprove. Our pure parray method can prove sat or unsat for all of them in 0.1s without any iteration. With our QElim method it is 0.14 second. In the `lastIndexOf` example, string s is searched for different non-overlapping substrings with a common long prefix (*e.g.* > 30 chars). As shown in [2], a BV method may iterate on the lengths many times before finding out a solution. Our pure parray method needs no iteration and solves all the cases in 0.5 second, while the best method in [2] takes about 15 seconds (note that their input substrings may be different from ours). Interestingly, this example can be solved in the automaton domain too, *i.e.* after the automaton refinement, valid solutions can be found in the automata. This takes 2.34 second in total. Clearly, automaton building and refinement incur extra overheads. The pure automaton method handles other examples poorly, hence the results are not shown.

For these examples, our QElim is not mandatory. But applying it can improve the solving performance, *e.g.* reduce the time of Example (a) from 7.3 second to 0.16 second. In addition, since Yices returns a partial model which may be incorrect, we need to double check this model when Yices's search is used. In the appendix we show some results comparing Yices's algorithm with ours. It is apparent that our QElim is essential in solving parray constraints.

Method	set 1 (70)		set 2 (30)	
	#solved	T.	#solved	T.
`Pure P-Array`	53	17.6s	0	–
+ our QElim	55	4.1s	21	18.5s
`Pure Automaton`	34	72s	3	2.6s
`P-Array+Autom.`	68	20.4s	3	2.6s
+ our QElim	70	6.3s	26	28s

Fig. 4. Evaluating various methods on Benchmark set I. #solved gives the number the solved cases (sat or unsat).

Benchmark set I consists of 100 tricky path conditions collected from (1) real Web applications, and (2) manual stress tests. It excludes those easy to solve. These benchmarks are divided into two sets: set 1 that Yices can handle forall constraints well, and set 2 where our QElim algorithm is required. We evaluate the solutions with pure parray, with pure automaton, or with parray+automaton, with results shown in Figure 4. The results also show the advantages of our method over automaton-based ones such as [9].

For set 1, Yices performs well, solving 55 cases quickly, but 2 of them are incorrect partial models. The other 15 involve regular expressions and needs the support of the automaton model, which allows the parray model to solve all

[2] Time comparison is rough due to different evaluation environments.

cases. However, using the pure automaton model only 34 cases can be solved since this model handles pure string constraints better than hybrid constraints.

Set 2 demonstrates the effectiveness of our QElim algorithm and automaton enhancement. Among these 30 cases, the "parray + automaton + QElim" method can solve 26, and the remaining 4 fails because the QElim algorithm hits the pre-defined limit (using a large limit can solve them). Missing any of these three components will lead to much inferior results, *e.g.* the pure parray method and pure automaton method can solve 0 and 3 respectively (time is counted for successful cases only). A closer look reveals that these two pure methods can prove unsat in short time, but are not good at finding solutions for sat cases. The combination of them does not help either since Yices seems to get stuck in handling the quantifiers.

Method Comparison. We compare PASS with two baseline implementations by us: an automaton-based method and a BV-like method . The former mimics the one in [9], with many details described in Section 4. The latter does not use bit-vectors directly. Instead *concrete* arrays are used to simulate bit-vectors with concrete lengths and indices. This can reuse our parray model and simplify the implementation: we first derive lengths constraints, then solve them to obtain length values, then instantiate the lengths in the forall constraints depicted in Tables 2 and 3, and then unroll all these constraints. Note that we do not directly compare PASS with existing tools since they assume different string operations and running settings, *e.g.* they accept much more restrictive syntax.

We run the three methods on several hundred non-trivial path conditions. Preliminary results indicate that PASS outperforms the simulated BV method in $\sim 80\%$ cases, and can be up to 150 times faster for some sat cases. For the rare cases where PASS performs worse, it is up to 4 times slower than other methods. Compared with the automaton-based method, PASS usually gains performance improvement in a magnitude of 1 or 2 orders, although the automaton-based method can sometimes (1) solve RegExp intensive path conditions faster, and (2) prove unsat faster. However, this comparison might be neither accurate nor conclusive since we simulate other methods and optimize PASS more. We leave the experimental comparison with external tools such as [8] and [7, 3] as future work.

6 Conclusions

We propose modeling strings with parameterized arrays, applying quantifier elimination to solve parray constraints, and using automata to model regular expressions and enhance the parray model. We show that all of these are essential to construct an efficient and comprehensive solver for numeric-string constraints with regular expressions.

The parray model needs much less enumeration of string lengths or values than existing methods since it encodes the string values and relations using quantified constraints of particular format. This format allows us to apply a simple algorithm to handle the quantifiers. Other enhancements are possible, *e.g.* more interactions between the parray and the automaton domains can further improve the performance.

References

1. Barrett, C., Conway, C. L., Deters, M., Hadarean, L., Jovanovic, D., King, T., Reynolds, A., and Tinelli, C. CVC4. In *CAV* (2011).
2. Bjørner, N., Tillmann, N., and Voronkov, A. Path feasibility analysis for string-manipulating programs. In *TACAS* (2009).
3. Bjørner, N., Tillmann, N., and Voronkov, A. Path feasibility analysis for string-manipulating programs. In *TACAS* (2009).
4. Bradley, A. R., Manna, Z., and Sipma, H. B. What's decidable about arrays? In *VMCAI* (2006).
5. Cadar, C., Dunbar, D., and Engler, D. R. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI* (2008).
6. Christensen, A. S., Møller, A., and Schwartzbach, M. I. Precise analysis of string expressions. In *SAS* (2003).
7. de Moura, L. M., and Bjørner, N. Z3: An efficient SMT solver. In *TACAS* (2008).
8. Ganesh, V., Kiezun, A., Artzi, S., Guo, P. J., Hooimeijer, P., and Ernst, M. D. HAMPI: A string solver for testing, analysis and vulnerability detection. In *CAV* (2011).
9. Ghosh, I., Shafiei, N., Li, G., and Chiang, W.-F. JST: An automatic test generation tool for industrial java applications with strings. In *ICSE* (2013).
10. Hooimeijer, P., and Veanes, M. An evaluation of automata algorithms for string analysis. In *VMCAI* (2011).
11. Hooimeijer, P., and Weimer, W. Solving string constraints lazily. In *ASE* (2010).
12. Li, G., Ghosh, I., and Rajan, S. P. KLOVER : A symbolic execution and automatic test generation tool for C++ programs. In *CAV* (2011).
13. Li, G., Li, P., Sawaga, G., Gopalakrishnan, G., Ghosh, I., and Rajan, S. P. GKLEE: Concolic verification and test generation for GPUs. In *PPoPP* (2012).
14. Li, G., and Slind, K. Trusted source translation of a total function language. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS)* (2008).
15. Saxena, P., Akhawe, D., Hanna, S., Mao, F., McCamant, S., and Song, D. A symbolic execution framework for JavaScript. In *S&P (Oakland)* (2010).
16. Shannon, D., Ghosh, I., Rajan, S., and Khurshid, S. Efficient symbolic execution of strings for validating web applications. In *2nd International Workshop on Defects in Large Software Systems* (2009).
17. Tillmann, N., and de Halleux, J. Pex-white box test generation for .net. In *TAP* (2008).
18. Veanes, M., de Halleux, P., and Tillmann, N. Rex: Symbolic regular expression explorer. In *ICST* (2010).
19. Yices: An SMT solver. `http://yices.csl.sri.com`.
20. Yu, F., Alkhalaf, M., and Bultan, T. Stranger: An automata-based string analysis tool for PHP. In *TACAS* (2010).
21. Yu, F., Bultan, T., and Ibarra, O. H. Symbolic string verification: Combining string analysis and size analysis. In *TACAS* (2009).