Strings at MOSCA

Matthew Hague[1]

Royal Holloway, University of London

Abstract

The first edition of MOSCA, the Meeting on String Constraints and Applications was held in May 2019 in the otherworldly location of Bertinoro [15]. For one week, 43 participants presented surveys and discussed recent research into the analysis of string constraints.

The following article is an overview of some of the topics discussed. It is broadly split into two sections: word equations and string constraints for verification. Presentations that did not fit neatly into this divide are summarised in a final section.

It is worth noting before beginning that the present survey is neither scientific nor systematic. The author is guilty of rambling at length on his familiar topics, and perhaps moving on too quickly when the content steps outside of his comfort zone. Thus, the world view presented is personal, and space allocated does not constitute a critical evaluation of the material.

1 Word Equations

1.1 Background

$$\text{there } X \text{ no there, } Y = Y \text{ is no } Y\text{, there}$$

Word equations are, on their surface, quite simple. In the above equation, a finite alphabet Σ is assumed, upper case letters are variables that can take string values from Σ^*, and lower case letters are constants. A solution to a word equation is any assignment of strings in Σ^* to the variables that makes the equation true. In the above case, assigning X to the string “is” and Y to the string “there” results in the required equality.

$$\text{there is no there, there} = \text{there is no there, there} .$$

The first-order theory of word equations was shown by Quine in 1946 to be undecidable [48] via the first-order theory of arithmetic. Similarly, Markov showed that word equations could be reduced to Hilbert’s 10th problem—integer solutions to Diophantine equations. However, he could only show decidability of word equations with two unknowns. Thus, the decidability of the general case, which would imply the undecidability of Diophantine equations, was left open.

Undecidability of Diophantine equations was eventually shown by Matiyasevich in 1970 [43] via the work of Davis, Putnam, and Robinson [17]. However, word equations remained stubbornly unsolved, with only further partial progress being made by Hmelevskii [28] in 1971 for the case of three unknowns. Seminal work in 1977 by Makanin finally showed the problem to be decidable [41].

Much work has followed, extending Makanin’s result to free groups [42], partially commutative monoids [44, 22], rational constraints [21], and so on.

However, the complexity of the original problem has remained a remarkable open problem. Makanin’s algorithm for word equations proceeds via a series of rewrites, with a large part of the difficulty lying in the proof of termination. The complexity of this algorithm has gradually been improved first to 4-NEXPTIME and then down to EXPSPACE [31, 50, 34, 34, 27].

Plandowski first showed in 1999 that a solution is possible in PSPACE [46]. This bound was improved recently by Jez to NSPACE($nlog(n)$), using a recompression technique [32]. The only known lower bound, however, is NP.

Two important extensions to standard word equations allow regular and length constraints. Regular constraints extend word equations with the ability to assert that the assignment to an unknown X must occur within some regular language $\mathcal{L}$. This variant was shown decidable by Schulz in 1990 [50]. Length constraints, however, are less straightforward. Let $|X|_a = |Y|_a$ assert that the number of "a" characters in the assignment to X matches that in the assignment to y. Word equations with such constraints are known to be undecidable [24, 8]. However, decidability when one may assert linear relations between the lengths of assignments, remains an important open problem.

1.2 Recompression and Complexity

Together with Rytter, Plandowski introduced the use of compression to the solving of word equations [47]. Using Lempel-Ziv compression (LZ77), length-minimal solutions can be compressed to a logarithmic of their original size. Using this technique, they were able to show a 2-NEXPTIME procedure. This was soon overtaken by Gutierrez's EXPSPACE variation of Makanin's algorithm, which does not use compression. Plandowski was later able to obtain the PSPACE algorithm mentioned above. However, this result also does not use compression so directly.

Recently, Jez revisited compression and was able to show an NSPACE($nlog(n)$) decision procedure. His approach utilizes a "local recompression" technique on which he had been working. A key difference in approach is that, while Plandowski's grammars work top-down, Jez's algorithm runs bottom-up.

Given a word

$$\mathrm{aaababcababbabcba}$$

we can first perform a simple compression of repeated letters

$$\mathrm{a_3babcabab_2abcba}\ .$$

Next, we can begin to identify repeated pairs of letters and replace them with a new letter. In our example

$$\mathrm{a_3bdcdd_2dce}$$

where

$$\begin{array}{lcl} \mathrm{d} & \rightarrow & \mathrm{ab} \\ \mathrm{e} & \rightarrow & \mathrm{ba}\ . \end{array}$$

As such, we are in effect compressing the solution by building a context free grammar. We iterate the above process, shrinking the size of the solution by a constant factor at each iteration, giving an exponential factor over n steps.

Unfortunately, it is problematic to run compression on a large pre-existing solution. Thus, it is natural to compress the equations, rather than the solutions. Such chicanery does not come for free: pairs of letters may not only appear inside the constant sections of the equation, but may cross the boundary between constants and variables. This problem is solved using "local decompression". Given a pair ab, this technique may replace a variable X with Xa, in effect pulling the a character out of the solution for X and into the constant part of the word equation. Thus Xb would become Xab. Compression can then proceed as before.

1.3 Quadratic Word Equations

1.3.1 Neilsen / Levi

In a quadratic word equation, each variable may be used at most twice. With such a restriction, Neilsen transformation or Levi's method gives a simple algorithm for decidability [23].

This method proceeds by searching a graph that can be built on-the-fly. Each node of the graph is a word equation, and the goal is to reach a node containing the trivial equality $\varepsilon = \varepsilon$. Edges of the graph are given by analysing prefixes. Suppose the current node contains the equation

$$XXYZ = \mathrm{ba}Y\mathrm{a}Z\mathrm{baa}\ .$$

In this case we immediately know that X must begin with ba. We perform a number of steps to construct the next node. First, we rewrite, uniformly, X to ba$X$ obtaining

$$\mathrm{ba}X\mathrm{ba}XYZ = \mathrm{ba}Y\mathrm{a}Z\mathrm{baa}\ .$$

Next the new node is obtained by cancelling the prefix ba from each side.

$$X\mathrm{ba}XYZ = Y\mathrm{a}Z\mathrm{baa}\ .$$

A key point is that the size of the equation does not increase. First we introduced a copy of the prefix ba before each occurrence of X, leading to an additional two copies of ba. However, we were also guaranteed to be able to cancel out two instances of ba, making the procedure neutral.

In the case where there is no prefix, we have a situation such as the

$$X\mathrm{ba}XYZ = Y\mathrm{a}Z\mathrm{baa}$$

above. In this case, there are three outgoing edges, for the following possible cases: either X and Y are equal, or X is a prefix of Y, or vice versa.

The simplest case is when they are equal. We can calculate the next node by first replacing all instances of Y with X

$$X\mathrm{ba}XXZ = X\mathrm{a}Z\mathrm{baa}$$

and then cancelling out the two X prefixes

$$\mathrm{ba}XXZ = \mathrm{a}Z\mathrm{baa}\ .$$

Note, although this at first doubles the number of X variables, the maximum number it can introduce is two. Since the cancellation step removes two Xs, the transformation is at worst neutral. In the example, observe that the branch cannot lead to a solution: the prefixes of the two sides cannot be reconciled.

For the remaining cases, we will suppose X is a prefix of Y. The other case is similar. When X is assumed to be a strict prefix, we replace Y with XY and obtain

$$X\mathrm{ba}XXYZ = XY\mathrm{a}Z\mathrm{baa}\ .$$

This effectively removes the X prefix from Y and makes it explicit in the equation. We can then cancel the preceding Xs

$$\mathrm{ba}XXYZ = Y\mathrm{a}Z\mathrm{baa}\ .$$

This gives us the next node. Note, this is again neutral, both in terms of size and in the number of X variables.

To implement the above procedure, one only needs PSPACE. If the trivial equation can be found, the original equation has a solution. If not, then the original equation has no solution.

For example, consider

$$XY = YX \ .$$

We take the branch assuming X equals Y and reach

$$XX = XX$$

which cancels to $X = X$. Cancelling prefixes again leads to $\varepsilon = \varepsilon$.

1.3.2 Length Constraints

Lin and Majumdar [38] have recently attempted to generalise the above procedure to handle quadratic word equations with length constraints. A standard approach to handling arithmetic constraints involves using the decidability of Presburger arithmetic. That is, boolean combinations of linear constraints. However, the following example begins to demonstrate that Presburger alone is not enough.

$$XY = YX$$

Solutions to the above equation can be shown to be of the form $X \in w^*$ and $Y \in w^*$ for some some word w. Notice, then, that the length of assignments to X and Y must both divide the length of w. Expanding on the above example, the equation

$$XabY = YabX$$

can be shown to have a length abstraction that can only be defined using a greatest-common-divisor operation. This is not Presburger definable.

Lin and Majumdar were able to define a class of quadratic word equations for which satisfiability is decidable. To do so, they following the Neilsen/Levi approach outlined above. The state machine defined by that approach was extended with counters that track the effect of each transition on the length of potential solutions to each variable. In the case when the associated counter system is “flat”—i.e. no nested loops—Presburger with divisibility is sufficient to define a length abstraction. Decidability then follows from the decidability of Presburger with divisibility constraints (of the form x divides y) [35]. Finally, it is conjectured that Presburger with divisibility is sufficient for the general case of quadratic word equations.

1.4 Fragments

Quadratic word equations are one fragment of general word equations. Many other such fragments can also be defined. Recent work by Day, Manea, and Nowotka has begun to explore such fragments to shed light on the long-standing gap between the NP lower-bound and PSPACE upper-bound [19]. Unfortunately, quadratic word equations have the same known complexities as the general case of word equations, despite a simpler PSPACE upper bound algorithm.

Strictly regular ordered word equations (SROWEs), however, can be shown to be NP-complete. Such a word equation is quadratic, with the additional constraint that every variable occurs once on each side of the equality, and in the same order. That is, an SROWE has the form

$$u_0 X_1 u_1 \dots X_n u_n = v_0 X_1 v_1 \dots X_n v_n \ .$$

In this case, it can be shown that solutions are always short, leading to a simple NP algorithm for deciding satisfiability. In fact, the complexity remains NP when additional regular, length, letter-counting, and subword constraints are allowed.

If the requirement that all variables appear on both sides is relaxed, it is no longer possible to allow additional constraints and remain in NP. By a reduction from the intersection of regular languages, one can show that non-strictly regular ordered word equations (ROWEs) with additional regular constraints is PSPACE-complete.

For the case of ROWEs without additional constraints, it is still possible to show NP-completeness of satisfiability. The algorithm, however, is quite involved and requires new tools regarding "chains" and "squares" of solutions.

Squares can also be used to identify another NP-complete fragment: that of regular reversed word equations. In these equations, the ordering is mirrored on either side of the equality.

1.5 Solving using SAT

The tool Woorpje uses SAT to solve word equations when the length of each string is known to be bounded [18]. With bounded lengths, there is a straight-forward encoding of the problem into a SAT solver: each position in the string can be encoded using a fixed number of boolean variables. However, this naive encoding does not work well. The contribution of Woorpje is to provide a more efficient encoding.

The encoding works by encoding, rather than the words directly, the run of an automaton that calculates solutions to the equation. This automaton intuitively guesses the solution and checks equality of both sides of the equation on a character by character basis. Because the length of any run is bounded (from the bound on solution lengths), a SAT formula can encode all runs.

Length abstraction is used to make the approach feasible. By abstracting each string by its length, constraints can be extracted. These constraints can be used to refine the upper bounds on the lengths of assignments to individual variables. MDDs (Multi-Decision Diagrams) are used to allow bounds to be refined, effectively on-the-fly, as the length of one string may affect the bounds of another.

2 Strings for Verification

A word equation is a restricted form of string constraint. One may also wish to allow logical connectives[1] and string relations other than concatenation and equality.

Recent years have seen a meteoric success in the development of constraint solvers. The unreasonable ability of SAT solvers to process extremely large sets of boolean constraints has been exploited by SMT solvers to handle constraints over a range of different theories. Typically, SMT solving has been used to tackle existential Presburger constraints. Current developments, however, focus on the addition of constraints over string variables.

String constraints turn out to be useful in several real-word applications.

2.1 Symbolic Execution

Symbolic execution [33, 12] attempts to explore as many paths of a program as possible, in the search for errors. To search all possible paths individually is clearly infeasible. Symbolic

[1]Satisfiability of a conjunction of word equations may be reduced to a single equation.

execution is effective because it does not explore concrete variable assignments, but instead explores paths symbolically. Consider the following toy program.

```
if (x > 0) {
    if (x < 0) {
        fail
    }
} else {
    if (x < 0) {
        fail
    }
}
```

This program has as many executions as there are allowed assignments to the variable `x`. However, a simple analysis reveals that there are only four true paths. Symbolic execution would treat the variable `x` as an unknown, and collect constraints on that variable along each syntactic path that it explores. In particular, the path to the first `fail` statement requires the constraints `x < 0` and `x > 0` to be satisfied by the value of `x`. A symbolic execution engine can forward the intersection of these constraints to a solver, which will reply that no such value of `x` exists. Thus, the `fail` can never be executed, and this path does not represent an error.

The path to the second `fail` statement collects the constraints `x <= 0` and `x < 0`. These two constraints can be simultaneously satisfied, and thus the path is an error. A symbolic execution tool can additionally generate a test case exercising this path using the model provided by the constraint solver.

Many symbolic execution engines exist, with Klee [9] being a well-known example. A symbolic version of the well-known Java Pathfinder tool, called Symbolic Pathfinder has recently been developed [45]. It supports multi-threading and string operations, and provides quantitative reasoning. That is, by counting the number of inputs that satisfy a certain condition, one can estimate the probability of hitting certain states. For this, we need to assume bounds on string length (otherwise there are infinite possibilities). Quantitative analysis has applications in security, such as side channel analysis and non-interference, and can be used to estimate how much information could be leaked. The MT-ABC tool, discussed in Section 2.3.3 provides solver-level support for quantitative analysis.

Recent work by Kinder *et al.* shows the importance of strings in the symbolic execution of JavaScript [40]. The contribution of Kinder *et al.*'s ExpoSE tool is the handling of JavaScript, and, in particular its prevalent use of strings and regular expressions. Of 400k JavaScript applications surveyed by Kinder *et al.*, 35% were found to make use of regular expressions.

String support in constraint satisfaction tools is in its infancy, and consequently Kinder *et al.* had to rely on symbolic execution of common string library functions such as "replace". In addition, there is a disconnect between regular expressions as discussed in theoretical computer science, and the "regexes" used in programming languages. For example, issues such as greediness, backreferences, and capture groups all required bespoke solutions to overcome the limitations of the underlying constraint solver. Thus, their work shows the importance of string support when analysing programs.

2.1.1 Access Control Rules

An example of string constraint solving used in large-scale industrial practice is that of access control rules at Amazon Web Services (AWS). Access control rules on a web service are used to determine who can or cannot access a resource, and the capabilities available to them. For example, a customer may be allowed to read a product listing, but should not be able to edit it. These rules are written by hand and can include tricky logical connectives such as negation as well as rules based on regular expressions. Hadarean discussed her work in the AWS team which focusses on ensuring that mistakes in these handwritten rules do not lead to the unintentional exposure of a vital resource to potentially malicious adversaries. Thus, analysis of strings satisfying regular expression constraints is important. A key feature of this problem is the need for rapid analysis, with responses expected within 20ms to meet the high demand of Amazon users.

2.2 Relations and Functions

Word equations use two kinds of string relation: equality and concatenation. In general, one may require string constraints using more relations. Relations may be defined in many ways. One particular representation of interest is that of "transducers". A transducer is a state machine that takes one word as input, and produces another as output. Although alternative models exist, such as streaming string transducers, we will take a simple base model.

A transducer over an alphabet Σ will consist of a finite set of states Q with an initial state q_I and accepting state q_F, and a transition relation $\Delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times (\Sigma \cup \{\varepsilon\}) \times Q$. We will write $q \xrightarrow[b]{a} q'$ for a transition $(q, a, b, q') \in \Delta$. We say a is the input character and b is the output. Let $q \xrightarrow[v]{u}^{*} q'$ denote a run: a sequence of transitions where the input word read is u and the output word produced is v. Note, because we allow ε in the input or output of a transition, it is possible for output to be produced without consuming input, and vice versa. We refer to these transducers as "asynchronous".

A transducer defines a relation $R \subseteq \Sigma^* \times \Sigma^*$ where $(u, v) \in R$ whenever $q_I \xrightarrow[v]{u}^{*} q_F$ is a run of the transducer.

For example, the transducer below simply rewrites the word $aabb$ to cc.

$$q_I \xrightarrow[\varepsilon]{a} q_1 \xrightarrow[c]{a} q_2 \xrightarrow[c]{b} q_3 \xrightarrow[\varepsilon]{b} q_F$$

In general, transducers may be used to represent transformations such as string sanitisation, HTML encoding, and substring extraction.

Note, the notion of transducers briefly defined above can be generalised to n-ary relations $R(w_1, \ldots, w_n)$.

2.2.1 Monadic Decomposition

Relations defined by transducers can often be difficult to work with as the input is closely tied to the output. Recognisable relations are a weaker notion of relation that is more amenable to decision procedures. A recognisable relation is defined as the finite union of products of regular languages. That is

$$R = \bigcup_{1 \leq i \leq n} \mathcal{L}_i \times \mathcal{L}'_i \ .$$

For example, the relation

$$R = \{(u, v) | u, v \in a^*\} \cup \{(u, v) | u, v \in b^*\}$$

is recognisable because

$$R = (a^* \times a^*) \cup (b^* \times b^*) \ .$$

The equality relation

$$R = \{(u, v) | u = v\}$$

however is not recognisable, but can be easily defined by a transducer.

When considering string constraints, it is easy to see the benefit of recognisable relations: they can be simply expressed using regular language containment and boolean connectives. That is $R(x, y)$ iff

$$\bigvee_i x \in \mathcal{L}_i \wedge y \in \mathcal{L}'_i \ .$$

Transforming a relation into its recognisable form is also known as "monadic decomposition". This term stems from the fact that, in the equation above, each clause only refers to a single variable, x or y. Thus, the variables may be treated separately.

Thus, determining whether a relation defined by a transducer is also recognisable can allow complete or more efficient decision procedures which may not exist in the general case. In the case of asynchronous transducers, it is known to be undecidable whether the defined relation can be monadically decomposed.

However, if the transducer is "synchronous"—i.e. ε does not appear in the transition relation—then the problem becomes decidable. In fact, even for the case of n-ary relations, the problem was recently shown by Barcelo *et al.* [6] to be PSPACE-complete for non-deterministic transducers, and NLOGSPACE-complete for deterministic ones.

2.2.2 Symbolic Transducers

The above transducer model works over a finite alphabet Σ. This is a standard assumption and works well for theoretical results. However, consider a transducer that reads a single character word, and replaces that character with an a. This transducer would require a single transition $q_I \xrightarrow[a]{b} q_F$ for each character $b \in \Sigma$. This is only a linear number of characters. However, the basic ASCII alphabet contains 128 characters, and modern text processing likely needs to support Unicode, which, in version 11.0, contains 137,374 characters. Working with individual characters, then, can slow down an algorithm by a factor between 128 and 137,374—this is quite a price to pay!

Motivated by such problems, D'Antoni and Veanes have studied "symbolic" automata and transducers [14, 51]. In this model, transitions are no longer labelled by characters, but are instead labelled by constraints over a given theory. Each transition would have an input variable i and an output variable o, and the constraints would assert properties of these two variables. In the case of the transducer that replaces a single character, we would only need one transition, $q_I \xrightarrow[o=a]{\top} q_F$. Note, the theory need not restrict itself to finite alphabets: the predicates may treat i and o as integer variables, for example.

This symbolic representation lends itself well to the representation of transductions. Often, real-world transductions shift characters. Consider a transition that replaces each lower case character with its upper case version. This can be compactly represented with two transitions: one for the case of lower case characters, and one for all other characters. Note, 97 is the ASCII code for 'a', 122 for 'z', and 32 is the distance between upper and lower case characters.

$$q \xrightarrow[o=i-32]{i\geq 97 \wedge i \leq 122} q' \qquad q \xrightarrow[o=i]{i<97 \vee i>122} q'$$

Similar transductions are highly important in security: input to programs often needs to be "sanitised" to remove malicious character combinations that may leads to injection attacks such as SQL injection and JavaScript injection.

When the theory used by the predicates is an effective boolean algebra, symbolic automata and transducers are generally well-behaved and have good closure and decidability properties. Extended symbolic automata and transducers, however, lack many desirable properties. These automata allow a certain amount of lookahead. That is, each predicate does not just refer to characters in the current transition, but may take into account the upcoming characters as well. In this case, many desirable properties can be regained with monadic decomposition. Similar to the monadic decomposition above, a predicate $\varphi(x_1, \ldots, x_n)$ admits a decomposition if it can be rewritten in as a finite union of the following form.

$$\bigvee_{1 \leq i \leq m} \varphi_i^1(x_1) \wedge \ldots \wedge \varphi_i^n(x_n)$$

If the predicate with lookahead can be rewritten to the above form, the automaton can be reorganised so that lookahead is no longer needed, recovering the required properties.

2.2.3 Parameterised Synchronicity

In the preceding discussion we touched upon synchronous and asynchronous transducers. In particular, while testing monadic decomposability of synchronous transducers is PSPACE-complete, it is undecidable for asynchronous transducers. Descotte, Figueira, and Puppis consider transducers where the synchronicity is parametric [20]. To explain, let a transition $q \xrightarrow[\varepsilon]{a} q'$ be represented by a 1, and a transition $q \xrightarrow[a]{\varepsilon} q'$ be represented by a 2. Thus, a 1 means an input character is read, and a 2 means an output character is produced. A synchronous transducer can be considered to be synchronised according to the pattern $(12)^*$. That is, the transducer reads an input, then produces an output. An asynchronous transducer, however, follows the most general synchronisation pattern $\{1, 2\}^*$. That is, at any moment, either an input will be read, or an output will be produced. The case of recognisable relations can also be represented using this idea, with the pattern 1^*2^*. This corresponds to a product $\mathcal{L}_1 \times \mathcal{L}_2$—first the input is read (1^*) and checked to be in $\mathcal{L}_1$, and then the output is read (2^*) and checked to be in $\mathcal{L}_2$.

Let $R(C)$ denote the relations definable by some transducer under pattern C. Descotte *et al.* show that it is decidable whether $R(C) \subseteq R(C')$ for given C and C'. It is also decidable whether $R(C)$ is closed under operations such as $\cup$, $\cap$, $\neg$, and $*$. Moreover, in some cases, given a relation $R \in R(C)$, it can be decided whether $R \in R(C')$.

2.2.4 Building Blocks of Regular Functions

Regular languages permit representations as algebra (regular expressions), state machines (automata), and logic (MSO). Motivated by the desire to replicate this elegant triangle for transducers, Dave, Gastin, and Krishna provide an algebraic representation of string functions [16]. In particular, they complete the triangle containing functional MSO transductions and either deterministic two-way transducers or streaming string transducers. The representation they consider is Regular Transducer Expressions (RTEs). Such expressions are built from the following grammar components.

- Constant strings w. Intuitively, given any input, the function defined by a constant w always returns w.
- If-then-else $\mathcal{L} ? E : E'$ given a regular language $\mathcal{L}$ and RTEs E and E'. If the given input belongs to w, then the output is $E(w)$, else it is $E'(w)$.

- Hadamard product $E \odot E'$ where the output on input w is $E(w) \cdot E'(w)$.
- Unambiguous Cauchy product $E \boxdot E'$ where the output on input w is $E(u) \cdot E'(v)$ whenever w can be uniquely factorised $w = u \cdot v$ where u is in the domain of E and v is in the domain of E'. The reverse operator is also permitted, where the output is $E'(v) \cdot E(u)$.
- Unambiguous Kleene-plus $E^{\boxplus}$ where the output on input w is $E(u_1) \cdot \cdots \cdot E(u_n)$ whenever w can be uniquely factorised $w = u_1 \cdot \cdots \cdot u_n$ where each $u_i$is in the domain of E. The reverse operator is also permitted, where the output is $E(u_n) \cdot \cdots \cdot E(u_1)$.
- Unambiguous 2-chained Kleene-plus $[\mathcal{L}, E]^{2\boxplus}$ where the output on input w is $E(u_1 u_2) \cdot \cdots \cdot E(u_{n-1} u_n)$ whenever w can be uniquely factorised $w = u_1 \cdot \cdots \cdot u_n$ where each $u_i$is in $\mathcal{L}$. The reverse operator is also permitted, where the output is $E(u_n u_{n-1}) \cdot \cdots \cdot E(u_2 u_1)$.

For infinite words, the Kleene operators may be replaced by similar ω-operators.

2.3 Solving String Constraints

String constraints can be formed from Boolean combinations of string relations (such as concatenation), string tests (such as containment in a regular language), and additional constraints (such as Presburger constraints over string lengths). The wide variety of possible string constraints, coupled with their frequent undecidability, has led to a number of string constraint solvers, each using different techniques, supporting different features, and with different guarantees on completeness. We will discuss a selection of the tools here that were discussed at MOSCA.

2.3.1 CVC4

CVC4 has been co-developed between Stanford and the University of Iowa [37]. It is a general purpose constraint solver supporting many theories, such as arithmetic, arrays, bit vectors, and strings with regular expressions.

Core Solver It builds upon the DPLL(T) architecture used by modern SMT solvers and provides a theory of strings that supports string variables, integer variables, string constants, string concatenation, string length, string equality, and linear arithmetic constraints. It is not known whether such constraints have a decidable satisfiability problem, but it has been observed that CVC4 runs well in practice.

The first step in such an SMT solver is to treat the problem as a basic SAT instance. Terms are treated as boolean values only, and a SAT solver looks for a satisfying assignment. For example, the constraint

$$x > 10 \wedge \neg(x > 10)$$

can easily be determined by a SAT solver to be unsatisfiable by treating $x > 10$ as the name of a boolean variable, rather than a term over the integers. However, the constraint

$$x \in a^* \wedge x \in b^*$$

cannot be determined unsatisfiable so directly. The SAT solver will notice that the formula can be satisfied by making both $x \in a^*$ and $x \in b^*$ true. An SMT solver will then ask a string theory solver whether these two constraints can be simultaneously satisfied. The string theory solver will reply negatively. The SMT solver will then add a lemma to block the assignment, and ask the SAT solver for another potential solution. In general, of course, this may not terminate.

In CVC4, string and integer constraints may be mixed, and hence the constraints are sent to both a string solver and an arithmetic solver. CVC4's arithmetic solver is based on standard techniques, such as the simplex method. Its string theory solver proceeds through five stages: length processing, congruence closure, normalisation of equalities, normalisation of disequalities, and finally cardinality.

In the length processing step, new integer constraints are inferred from the string constraints which can be sent to the arithmetic solver. For example, the length of the string constant bba can be fixed to 3, and, a clause $x = y \cdot z$ implies that the length of x equals the sum of the lengths of y and z.

Next, the congruence closure step groups terms into equivalence classes. For example $x = \mathrm{a} \cdot y$ has the equivalence classes $\{x, \mathrm{a} \cdot y\}, \{\mathrm{a}\}, \{y\}$. A conflict can be returned if terms asserted to be unequal find themselves in the same equivalence class.

The normalisation step on equality iteratively replaces terms with a representative from the equivalence class it occupies, and then recomputes the congruence closure. This is not always straightforward. For example given an equivalence class $\{x \cdot y, x' \cdot y\}$ the alignment between x, y, x', and y' contains several cases, and may require the splitting of equivalence representatives. Selecting alignments and splits can be crucial to performance.

The normalisation step on disequalities proceeds analogously, except the alignment is aiming to find pairs that are not equal, instead of equal.

Finally, the cardinality step makes inferences on the number of possible strings given any length bounds and the size of the alphabet. A simple example is if the alphabet is size 3, but a solution would require 4 distinct strings of length 1. In this case, the solution can be discarded.

Extended String Constraints In practice, one often needs to make use of string relations such as substr9x, 0, "a") (that asserts that x contains "a" as a substring starting from position 0), contains(x, "a") (that asserts that x contains "a" as a substring), or replace(x, "a", "b") (that replaces the first occurrence of a string with another).

Both CVC4 and Z3 approach these constraints using a preprocessing step. The idea is to rewrite these extended terms into the core language. The relation ¬contains(x, "a") is approached by CVC4 by using a bounded ∀ and substr constraints

$$\forall i < |x|. \neg \mathrm{substr}(x, i, \text{“a”}) \ .$$

Then, if we make the assumption that x has length at most 5, we can replace the ∀ with a conjunction

$$\neg \mathrm{substr}(x, 0, \text{“a”}) \wedge \cdots \wedge \neg \mathrm{substr}(x, 3, \text{“a”}) \ .$$

Finally, substr can be replaced by concatenation. That is substr(x, i, w) becomes

$$x = u \cdot w \cdot v \wedge |u| = i \ .$$

Such an approach in incomplete and requires a good set of heuristics. Solvers such as CVC4 and Z3 make use of techniques such as context-dependent simplification and lazy expansion to ensure these rewriting steps can often lead to success.

2.3.2 Z3 and Z3str

There are two forms of support for strings in Z3. The first is the built into standard Z3, while the second is known as Z3str.

Z3 Z3 has supported strings since 2009 [7] using the observation that bounded length strings can be supported as arrays. The string support maintains a state machine containing current solutions to variables, unsolved (dis-)equalities, unsolved "contains" clauses, and unsolved automata memberships. The search for a solution proceeds via a number of steps including rewriting to a normal form, solving when lengths are known to be bounded, and case splitting along potential concatenation points.

Z3str Z3str [53] uses an automata-based approach (see next section). It supports linear arithmetic over string length, membership in regular languages, string-to-number conversion, and concatenation. It's algorithm is based on a DPLL(T) procedure. Without string-to-number conversion, the theory is known to be decidable. When string-to-number conversion is allowed, the theory becomes undecidable.

A key insight that it uses is that formulas often contain both implicit and explicit length information, and that this information can be used to speed up the search for a solution. The solver analyses the current regular language constraints over the variables and maintains a lower and upper bound estimate of the string length. When these bounds determine the length of the string, the question can be replaced with character constraints. For variables without tight bounds, the solver will look for variables constrained by multiple languages. Using an estimation of the work required, decide the perform some of the language intersections, in the hope of tightening some bounds.

2.3.3 Automata-Based Solvers

CVC4 supports a basic core fragment and attacks complex string constraints using a rewriting phase. An alternative approach is to represent string relations as transducers and take an automata-based approach to the constraint solving problem. We overview several approaches discussed at MOSCA.

SLOTH The SLOTH tool is based on the use of alternating automata [29]. It supports "straight-line" conjunctions of string constraints built from regular constraints, concatenation, and transductions.

The straight-line restriction intuitively corresponds to single-static assignment form. Each term is of the form

$$x = f(y_1, \ldots, y_n) \ .$$

In an equality $x = y \cdot z$ the value of x depends on the value of y and z. In the straight-line fragment, these dependencies must not be circular. A non-straight line constraint is

$$x = y \cdot z \wedge y = x \cdot z$$

in which x depends on y and vice-versa. Note, the given constraint may be satisfied by assigning all strings to empty.

The ability to support transductions allows relations such as substr(x, i, w), contains(x, w), and replace(x, w, w') to be encoded. Moreover, constraints received from symbolic execution engines are likely to satisfy the straight-line requirement, since they correspond to program paths. The DPLL(T) framework can additionally be used to provide some support for disjunction. Thus, the straight-line fragment can be quite useful in practice.

The fragment considered by SLOTH is known to be decidable [39]. Thus, SLOTH provides completeness guarantees not provided by CVC4. However, it can be observed that CVC4 is much faster on the constraints it supports.

Alternating automata are used as an underlying representation of sets of strings by SLOTH. In a standard finite automaton, pairs of transitions of the form $q \xrightarrow{a} q_1$ and $q \xrightarrow{a} q_2$ can be considered to be disjunctions: the word is accepted if either the first transition leads to an accepting run, or the second does. In an alternating automaton, states can be considered existential—corresponding to the aforementioned disjunctive interpretation—or universal. From a universal state, such a pair of transitions would require both the first and the second transition to lead to an accepting run.

The use of alternation leads to exponentially more succinct automata, with a corresponding increase in the complexity of emptiness checking. While a standard finite automaton can be checked in LOGSPACE, an alternating automaton requires PSPACE.

In order to make use of alternating automata, SLOTH develops a notion of alternating transducers, and devises new algorithms for combining constraints into a single automaton, and then efficiently checking emptiness.

OSTRICH The OSTRICH tool [10] deals with a similar straight-line fragment to SLOTH. Its basic algorithm is a generic algorithm supporting any string functions that can be shown to satisfy certain conditions. Transducers and concatenation have these properties, as does the more complex replaceall(x, w, y) function, where all instances of w in x are replaced by the contents of the variable y. This goes beyond transducers in which y must be a constant value.

The condition the relations must satisfy is most easily explained for a term $x = f(y)$. Suppose we have inferred a regular constraint $y \in \mathcal{L}$. We require $f^{-1}(\mathcal{L})$ to also be a regular language. That is, a regular constraint on y can be pulled backwards through f to form a regular constraint on x. In this way, relations can be eliminated, as shown in the following example. Beginning with a constraint

$$y = f(x) \wedge z = f'(y) \wedge z \in \mathcal{L}$$

we can pull $\mathcal{L}$ backwards through f' to obtain

$$y = f(x) \wedge y \in \mathcal{L}'$$

for some $\mathcal{L}'$. We do this again to obtain

$$x \in \mathcal{L}'' \ .$$

Finding a satisfying assignment means testing $\mathcal{L}''$ for emptiness.

In the general case, string functions may take several arguments. In this case, we require recognisable relations. Recall, these are finite unions of products of languages

$$\bigcup_i \mathcal{L}_1^i \times \cdots \times \mathcal{L}_n^i \ .$$

Given a term

$$y = f(x_1, \ldots, x_n)$$

and a regular constraint $y \in \mathcal{L}$, we require $f^{-1}(\mathcal{L})$ to be a recognisable relation. If this condition is satisfied by all functions, the algorithm demonstrated above can be used to eliminate string functions.

The current version of OSTRICH only provides primitive support for integer or length constraints. Recent, unpublished, work by Zhilin Wu has extended the automata representation to "integer cost register automata". Using these automata, it becomes possible to analyse constraints over string lengths and the positions of substrings in words. In order to maintain decidability, string/integer functions need to preserve the automaton representation just as in the case with only string.

Intersections of Regular Constraints The OSTRICH algorithm can be considered a reduction algorithm: constraints over a general set of terms are reduced to the intersection of regular constraints. Unfortunately, OSTRICH's algorithm for solving these intersections is a simple on-the-fly product construction.

Recently, Cox and Leasure have studied the corpus problem [13]. In this problem, we are given a corpus of regular expressions, and a candidate new regular expression. The expression should be added to the corpus if it accepts a string that is not already accepted by an existing expression. That is, for a regular language $\mathcal{L}$ and existing languages $\mathcal{L}_1, \ldots, \mathcal{L}_n$ we are interested in the satisfiability of

$$x \in \mathcal{L} \wedge x \notin \mathcal{L}_1 \wedge \cdots \wedge x \notin \mathcal{L}_n \ .$$

Instances of these problems were tested on CVC4, Z3, SLOTH, and OSTRICH, and all tools showed exponential behaviour on growing problem sizes. To address this problem, the Qzy tool was built, which shows only quadratic scaling in the number of regular expressions. Qzy uses the observation that the corpus problem is really a safety-checking problem over a transition system (or, indeed, several transition systems running in parallel). Thus, the problem can be tackled using a model checker, implementing an algorithm such as IC3, with several additional bespoke optimisations.

The study of language inclusion problems and the connection to model checking techniques, such as anti-chains, has been well studied. The Language Inclusion project collects a large body of work on this topic [2].

Circuit-based Solving The SLOG and SLENT tools [52] also use model checking techniques to handle the analysis of string constraints based on automata. SLENT is an extension of SLOG, which provides support for string lengths. The idea of the approach is to use a circuit-based representation of automata, where boolean formulas are used to represent the initial states, final states, and transition relation. Circuits can additionally be used to encode intersection, union, and string concatenation operations. In addition, replacement, reverse, and prefix and suffix operations are supported. The resulting circuit is then analysed using Property Directed Reachability techniques that have been successful in hardware and software model checking.

An additional benefit of this approach is the ability to generate filters. Filters are motivated by the application of string constraint solving to input santisation. In situations where suitably crafted malicious inputs may cause undesired behaviour, a filter can be used to identify in advance whether a given input may be malicious, and prevent the program running on it.

Multi-track Automata and Model Counting In some cases, satisfiability alone is not enough. For example, if it were required to quantify the probability of an error occurring, information about the number of possible inputs that could lead to an error is required. Aydin *et al.* [5] have developed the tool MT-ABC, which is a model-counting string constraint solver.

For quantitative analysis, the number of models is counted with respect to a bound on the string length. This avoids the problem of an infinite number of solutions.

The tool supports a large range of string functions, including length constraints. It uses automata to represent sets of strings. By constructing an automaton that represents the set of strings satisfying the constraint, model counting reduces to counting the number of accepting paths up to the given bound.

To faithfully capture relationships between variables, it is not enough to use one automaton per program variable. For example, equal(x, y) cannot be represented with two automata: one for x and one for y; both variables need to be considered simultaneously. For this reason, multi-track automata are used.

Using this representation, the MT-ABC tool is able to take a constraint as input and produce as output a function $f : \mathbb{N} \to \mathbb{N}$. This function takes as an argument a bound on the length of string variables, and returns the number of solutions to the original constraint whose length does not exceed the given bound.

Beyond Regular Constraints Abdulla *et al.* [1] go beyond the string constraints considered so far. In addition to equalities and disequalities (word equations), transductions, and length constraints, they allow membership tests of context-free grammars. That is, $x \in \mathcal{L}$ where $\mathcal{L}$ is no longer regular, but context-free. Such constraints are clearly undecidable (as conjunctions of context-free grammars have an undecidable emptiness problem). To tackle this problem, the authors introduce a "flatten and conquer" framework, which uses a counter-example guided abstraction-refinement loop, that utilises both under- and over-approximation.

Flattening is with with respect to an "abstraction parameter", which is a pair (n, m) of natural numbers. The parameter limits the search for solutions to those strings of the form $w_1^* \ldots w_n^*$ where each w_i is of length at most m. Checking intersection of context-free languages with respect to such languages (known as "bounded languages") can be reduced to Presburger constraint solving [25].

The refinement loop proceeds by first using any over-approximation to find a potentially spurious satisfying assignment. From this assignment, the set of abstraction parameters that could have generated that word are added to a "waiting" set.

Next, the under-approximation phase iterates through each of the waiting abstraction parameters, and tests whether there exists a solution with respect to the appropriate flattening. If a solution exists, it will be genuine. If no solution is found, over-approximation is used to find more potential abstraction parameters. This loop may not terminate, but can be effective. It is implemented in the open-source TRAU tool.

2.4 Constraint Programming

In this section we have considered string constraints from an SMT perspective, based on SAT solving. SAT solving may also be generalised to the area of constraint programming, where constraints are considered over arbitrary, but finite domains. Primarily, finding solutions to constraint programs is based on intelligent backtracking and propagation algorithms, with some recent uses of clause learning. The use of global constraints, such as AllDifferent means propagation is often more advanced that the unit propagation typically used in SAT solving.

The domain of strings has been a consideration in constraint programming. In order to study a finite domain, strings are bounded up to a given length b. These strings may be handled in a number of ways. One might represent strings as arrays of characters, with a suitable padding character for strings shorter than the bound b. Additionally, one may implement bespoke propagation algorithms that treat the padding character as special. Alternatively, a special string data-type may be added. When such a type is added, there is a choice of representation and the interaction with the propagation loop.

One possible representation is that of "dashed strings". In this representation, a set of strings is represented as a sequence of sets of string which may be repeated to form a string of length given by an interval. For example

$$\{\mathrm{B}, \mathrm{b}\}^{1,2}\{\mathrm{ac}, \mathrm{dc}\}^{3,8}$$

represents strings with one or two B or b characters, following by a three to eight character string formed by concatenations of ac and dc.

This approach has been shown to be competitive, and has been implemented in the MiniZinc tool [3].

3 Omissions

We have discussed a range of topics covered by presentations at MOSCA 2019. Inevitably, there are some that have been omitted that do not fit the main threads focussed on by the present article. We briefly mention these here.

Strings are closely related to the topic of graph databases, which were ably summarised in a survey presentation by Barcelo [4]. Similarly, regular model checking reduces many model checking problems to the analysis of string representations of program configurations. This topic was surveyed by Vojnar, its applications to probabilistic concurrent systems discussed by Lengal [36], and extensions beyond safety by Hong [30].

Chistikov discussed a word problem related to the balancing of well-bracketed words [11] while Piskac presented recent work into synthesis, showing that programming by example, while fast, was not always considered helpful by its users [49]. Lengal also discussed work attempting to improve on MONA for the analysis WS1S, which can be interpreted as a string logic [26].

Finally, the MOSCA meeting hosted a working session on SMT-LIB for strings—a standard that is currently under active development.

References

[1] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Bui Phi Diep, Lukás Holík, Ahmed Rezine, and Philipp Rümmer. Flatten and conquer: a framework for efficient analysis of string constraints. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 602–617, 2017.

[2] Parosh Aziz Abdulla, Yu-Fang Chen, Lorenzo Clemente, Seth Fogarty, Lukas Holik, Chih-Duo Hong, Ondra Lengal, Richard Mayr, Jiri Simacek, Moshe Vardi, and Tomas Vojnar. `http://languageinclusion.org`, 2017. [Online; accessed 18-July-2019].

[3] Roberto Amadini, Pierre Flener, Justin Pearson, Joseph D. Scott, Peter J. Stuckey, and Guido Tack. Minizinc with strings. In *Logic-Based Program Synthesis and Transformation - 26th International Symposium, LOPSTR 2016, Edinburgh, UK, September 6-8, 2016, Revised Selected Papers*, pages 59–75, 2016.

[4] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan L. Reutter, and Domagoj Vrgoc. Foundations of modern query languages for graph databases. *ACM Comput. Surv.*, 50(5):68:1–68:40, 2017.

[5] Abdulbaki Aydin, William Eiers, Lucas Bang, Tegan Brennan, Miroslav Gavrilov, Tevfik Bultan, and Fang Yu. Parameterized model counting for string and numeric constraints. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, pages 400–410, 2018.

[6] Pablo Barceló, Chih-Duo Hong, Xuan Bach Le, Anthony W. Lin, and Reino Niskanen. Monadic decomposability of regular relations. In *46th International Colloquium on Automata, Languages, and Programming, ICALP 2019, July 9-12, 2019, Patras, Greece.*, pages 103:1–103:14, 2019.

[7] Nikolaj Bjørner, Nikolai Tillmann, and Andrei Voronkov. Path feasibility analysis for string-manipulating programs. In *Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on*

Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings, pages 307–321, 2009.

[8] J Richard Büchi and Steven Senger. Definability in the existential theory of concatenation and undecidable extensions of this theory. In *The Collected Works of J. Richard Büchi*, pages 671–683. Springer, 1990.

[9] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[10] Taolue Chen, Matthew Hague, Anthony W. Lin, Philipp Rümmer, and Zhilin Wu. Decision procedures for path feasibility of string-manipulating programs with complex operations. *PACMPL*, 3(POPL):49:1–49:30, 2019.

[11] Dmitry Chistikov and Mikhail Vyalyi. Re-pairing brackets. *CoRR*, abs/1904.08402, 2019.

[12] Lori A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Trans. Software Eng.*, 2(3):215–222, 1976.

[13] Arlen Cox and Jason Leasure. Model checking regular language constraints. *CoRR*, abs/1708.09073, 2017.

[14] Loris D'Antoni and Margus Veanes. The power of symbolic automata and transducers. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I*, pages 47–67, 2017.

[15] Loris D'Atoni, Anthony W. Lin, and Philipp Rümmer. Meeting on string constraints and applications. `https://mosca19.github.io`, 2019.

[16] Vrunda Dave, Paul Gastin, and Shankara Narayanan Krishna. Regular transducer expressions for regular transformations. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, pages 315–324, 2018.

[17] Martin Davis, Hilary Putnam, and Julia Robinson. The decision problem for exponential diophantine equations. *Annals of Mathematics*, 74(3):425–436, 1961.

[18] Joel D. Day, Thorsten Ehlers, Mitja Kulczynski, Florin Manea, Dirk Nowotka, and Danny Bøgsted Poulsen. On solving word equations using SAT. In *Reachability Problems - 13th International Conference, RP 2019, Brussels, Belgium, September 11-13, 2019, Proceedings*, pages 93–106, 2019.

[19] Joel D. Day, Florin Manea, and Dirk Nowotka. The hardness of solving simple word equations. In *42nd International Symposium on Mathematical Foundations of Computer Science, MFCS 2017, August 21-25, 2017 - Aalborg, Denmark*, pages 18:1–18:14, 2017.

[20] María Emilia Descotte, Diego Figueira, and Gabriele Puppis. Resynchronizing classes of word relations. In *45th International Colloquium on Automata, Languages, and Programming, ICALP 2018, July 9-13, 2018, Prague, Czech Republic*, pages 123:1–123:13, 2018.

[21] Volker Diekert, Claudio Gutiérrez, and Christian Hagenah. The existential theory of equations with rational constraints in free groups is pspace-complete. In *STACS 2001, 18th Annual Symposium on Theoretical Aspects of Computer Science, Dresden, Germany, February 15-17, 2001, Proceedings*, pages 170–182, 2001.

[22] Volker Diekert, Yuri V. Matiyasevich, and Anca Muscholl. Solving trace equations using lexicographical normal forms. In *Automata, Languages and Programming, 24th International Colloquium, ICALP'97, Bologna, Italy, 7-11 July 1997, Proceedings*, pages 336–346, 1997.

[23] Volker Diekert and John Michael Robson. Quadratic word equations. In *Jewels are Forever, Contributions on Theoretical Computer Science in Honor of Arto Salomaa*, pages 314–326, 1999.

[24] Valery Durnev. Studying algorithmic problems for free semi-groups and groups. In *Logical Foundations of Computer Science, 4th International Symposium, LFCS'97, Yaroslavl, Russia, July 6-12, 1997, Proceedings*, pages 88–101, 1997.

[25] Javier Esparza and Pierre Ganty. Complexity of pattern-based verification for multithreaded

programs. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 499–510, 2011.

[26] Tomás Fiedor, Lukás Holík, Petr Janku, Ondrej Lengál, and Tomás Vojnar. Lazy automata techniques for WS1S. In *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I*, pages 407–425, 2017.

[27] Claudio Gutiérrez. Satisfiability of word equations with constants is in exponential space. In *39th Annual Symposium on Foundations of Computer Science, FOCS '98, November 8-11, 1998, Palo Alto, California, USA*, pages 112–119, 1998.

[28] Ju. I. Hmelevskii. Equations in free semigroups. *Trudy Mat. Inst. Steklov.*, 107, 1971.

[29] Lukás Holík, Petr Janku, Anthony W. Lin, Philipp Rümmer, and Tomás Vojnar. String constraints with concatenation and transducers solved efficiently. *PACMPL*, 2(POPL):4:1–4:32, 2018.

[30] Chih-Duo Hong, Anthony W. Lin, Rupak Majumdar, and Philipp Rümmer. Probabilistic bisimulation for parameterized systems - (with applications to verifying anonymous protocols). In *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*, pages 455–474, 2019.

[31] Joxan Jaffar. Minimal and complete word unification. *J. ACM*, 37(1):47–85, 1990.

[32] Artur Jez. Recompression: a simple and powerful technique for word equations. In *30th International Symposium on Theoretical Aspects of Computer Science, STACS 2013, February 27 - March 2, 2013, Kiel, Germany*, pages 233–244, 2013.

[33] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[34] Antoni Koscielski and Leszek Pacholski. Complexity of makanin's algorithm. *J. ACM*, 43(4):670–684, 1996.

[35] Antonia Lechner, Joël Ouaknine, and James Worrell. On the complexity of linear arithmetic with divisibility. In *30th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2015, Kyoto, Japan, July 6-10, 2015*, pages 667–676, 2015.

[36] Ondrej Lengál, Anthony Widjaja Lin, Rupak Majumdar, and Philipp Rümmer. Fair termination for parameterized probabilistic concurrent systems. In *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I*, pages 499–517, 2017.

[37] Tianyi Liang, Andrew Reynolds, Nestan Tsiskaridze, Cesare Tinelli, Clark W. Barrett, and Morgan Deters. An efficient SMT solver for string constraints. *Formal Methods in System Design*, 48(3):206–234, 2016.

[38] Anthony W. Lin and Rupak Majumdar. Quadratic word equations with length constraints, counter systems, and presburger arithmetic with divisibility. In *Automated Technology for Verification and Analysis - 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7-10, 2018, Proceedings*, pages 352–369, 2018.

[39] Anthony Widjaja Lin and Pablo Barceló. String solving with word equations and transducers: towards a logic for analysing mutation XSS. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 123–136, 2016.

[40] Blake Loring, Duncan Mitchell, and Johannes Kinder. Sound regular expression semantics for dynamic symbolic execution of javascript. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019.*, pages 425–438, 2019.

[41] G. S. Makanin. The problem of solvability of equations in a free semigroup. *Sbornik: Mathematics*, 32, 2:129–198, 1977.

[42] G. S. Makanin. Equations in a free group. *Izv. Akad. Nauk SSSR, Ser. Mat.*, 46:1199–1273, 1982.

[43] Yu. Matiyasevich. The Diophantineness of enumerable sets. *Dokl. Akad. Nauk SSSR*, 191:279–282, 1970.

[44] Yuri V. Matiyasevich. Some decision problems for traces. In *Logical Foundations of Computer Science, 4th International Symposium, LFCS'97, Yaroslavl, Russia, July 6-12, 1997, Proceedings*, pages 248–257, 1997.

[45] Yannic Noller, Corina S. Pasareanu, Aymeric Fromherz, Xuan-Bach D. Le, and Willem Visser. Symbolic pathfinder for SV-COMP - (competition contribution). In *Tools and Algorithms for the Construction and Analysis of Systems - 25 Years of TACAS: TOOLympics, Held as Part of ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part III*, pages 239–243, 2019.

[46] Wojciech Plandowski. Satisfiability of word equations with constants is in PSPACE. In *40th Annual Symposium on Foundations of Computer Science, FOCS '99, 17-18 October, 1999, New York, NY, USA*, pages 495–500, 1999.

[47] Wojciech Plandowski and Wojciech Rytter. Application of lempel-ziv encodings to the solution of words equations. In *Automata, Languages and Programming, 25th International Colloquium, ICALP'98, Aalborg, Denmark, July 13-17, 1998, Proceedings*, pages 731–742, 1998.

[48] W. V. Quine. Concatenation as a basis for arithmetic. *J. Symb. Log.*, 11(4):105–114, 1946.

[49] Mark Santolucito, Drew Goldman, Allyson Weseley, and Ruzica Piskac. Programming by example: Efficient, but not "helpful". In *9th Workshop on Evaluation and Usability of Programming Languages and Tools, PLATEAU@SPLASH 2018, November 5, 2018, Boston, Massachusetts, USA*, pages 3:1–3:10, 2018.

[50] Klaus U. Schulz. Makanin's algorithm for word equations - two improvements and a generalization. In *Word Equations and Related Topics, First International Workshop, IWWERT '90, Tübingen, Germany, October 1-3, 1990, Proceedings*, pages 85–150, 1990.

[51] Margus Veanes. Symbolic automata theory with applications (invited talk). In *26th EACSL Annual Conference on Computer Science Logic, CSL 2017, August 20-24, 2017, Stockholm, Sweden*, pages 7:1–7:3, 2017.

[52] Hung-En Wang, Shih-Yu Chen, Fang Yu, and Jie-Hong R. Jiang. A symbolic model checking approach to the analysis of string and length constraints. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, pages 623–633, 2018.

[53] Yunhui Zheng, Vijay Ganesh, Sanu Subramanian, Omer Tripp, Murphy Berzish, Julian Dolby, and Xiangyu Zhang. Z3str2: an efficient solver for strings, regular expressions, and length constraints. *Formal Methods in System Design*, 50(2-3):249–288, 2017.

A Participants

- Mohammed Faouzi Atig, Uppsala University, Sweden
- Pablo Barcelo, University of Chile, Chile
- Murphy Berzish, University of Waterloo, Canada
- Nikolaj Bjorner, Microsoft Research, USA
- Yu-Fang Chen, Academia Sinica, Taiwan
- Dmitry Chistikov, University of Warwick, UK
- Arlen Cox, IDA/CCS, USA
- Joel Day, Loughborough University, UK

- Vrunda Dave, IIT Bombay, India
- Volker Diekert, Universität Stuttgart, Germany
- Bui Phi Diep, Uppsala University, Sweden
- Diego Figueira, LaBRI, France
- Liana Hadarean, Amazon, USA
- Matthew Hague, Royal Holloway, UK
- Lukas Holik, Brno University of Technology, Czechia
- Chih-Duo Hong, Oxford University, UK
- Joxan Jaffar, National University of Singapore, Singapore
- Petr Janku, Brno University of Technology, Czechia
- Artur Jez, University of Wroclaw, Poland
- Roland Jiang, National Taiwan University, Taiwan
- Johannes Kinder, Bundeswehr University Munich, Germany
- Mitja Kulczynski, Kiel University, Germany
- Quang Loc Le, Teeside University, UK
- Ondrej Lengal, Brno University of Technology, Czechia
- Anthony W. Lin, Oxford University, UK
- Rupak Majumdar, MPI-SWS, Germany
- Richard Mayr, Edinburgh University, UK
- Duncan Mitchell, Royal Holloway, UK
- Muhammad Najib, Oxford University, UK
- Reino Niskanen, Oxford University, UK
- Dirk Nowotka, Kiel University, Germany
- Corina Pasareanu, NASA, USA
- Justin Pearson, Uppsala University, Sweden
- Ruzica Piskac, Yale University, USA
- Andrew Reynolds, University of Iowa, USA
- Philipp Rümmer, Uppsala University, Sweden
- Krishna S., IIT Bombay, India
- Cesare Tinelli, University of Iowa, USA

- Nestan Tsiskaridze, University of California Santa Barbara, USA
- Margus Veanes, Microsoft Research, USA
- Tomas Vojnar, Brno University of Technology, Czechia
- Zhilin Wu, Chinese Academy of Science, China
- Fang Yu, National Chengchi University, Taiwan