String Constraints with Concatenation and Transducers Solved Efficiently

LUKÁŠ HOLÍK, Brno University of Technology, Czech Republic
PETR JANKŮ, Brno University of Technology, Czech Republic
ANTHONY W. LIN, University of Oxford, United Kingdom
PHILIPP RÜMMER, Uppsala University, Sweden
TOMÁŠ VOJNAR, Brno University of Technology, Czech Republic

String analysis is the problem of reasoning about how strings are manipulated by a program. It has numerous applications including automatic detection of cross-site scripting, and automatic test-case generation. A popular string analysis technique includes symbolic executions, which at their core use constraint solvers over the string domain, a.k.a. string solvers. Such solvers typically reason about constraints expressed in theories over strings with the concatenation operator as an atomic constraint. In recent years, researchers started to recognise the importance of incorporating the replace-all operator (i.e. replace all occurrences of a string by another string) and, more generally, finite-state transductions in the theories of strings with concatenation. Such string operations are typically crucial for reasoning about XSS vulnerabilities in web applications, especially for modelling sanitisation functions and implicit browser transductions (e.g. innerHTML). Although this results in an undecidable theory in general, it was recently shown that the straight-line fragment of the theory is decidable, and is sufficiently expressive in practice. In this paper, we provide the first string solver that can reason about constraints involving both concatenation and finite-state transductions. Moreover, it has a completeness and termination guarantee for several important fragments (e.g. straight-line fragment). The main challenge addressed in the paper is the prohibitive worst-case complexity of the theory (double-exponential time), which is exponentially harder than the case without finite-state transductions. To this end, we propose a method that exploits succinct alternating finite-state automata as concise symbolic representations of string constraints. In contrast to previous approaches using nondeterministic automata, alternation offers not only exponential savings in space when representing Boolean combinations of transducers, but also a possibility of succinct representation of otherwise costly combinations of transducers and concatenation. Reasoning about the emptiness of the AFA language requires a state-space exploration in an exponential-sized graph, for which we use model checking algorithms (e.g. IC3). We have implemented our algorithm and demonstrated its efficacy on benchmarks that are derived from cross-site scripting analysis and other examples in the literature.

CCS Concepts: • **Theory of computation → Automated reasoning**; **Verification by model checking**; **Program verification**; **Program analysis**; *Logic and verification*; Complexity classes;

Authors' addresses: Lukáš Holík, Brno University of Technology, Faculty of Information Technology, IT4Innovations Centre of Excellence, Božetěchova 2, Brno, CZ-61266, Czech Republic, holik@fit.vutbr.cz; Petr Janků, Brno University of Technology, Faculty of Information Technology, IT4Innovations Centre of Excellence, Božetěchova 2, Brno, CZ-61266, Czech Republic, ijanku@fit.vutbr.cz; Anthony W. Lin, Department of Computer Science, University of Oxford, Wolfson Building, Parks Road, Oxford, OX1 3QD, United Kingdom, anthony.lin@cs.ox.ac.uk; Philipp Rümmer, Department of Information Technology, Uppsala University, Box 337, Uppsala, 75105, Sweden, philipp.ruemmer@it.uu.se; Tomáš Vojnar, Brno University of Technology, Faculty of Information Technology, IT4Innovations Centre of Excellence, Božetěchova 2, Brno, CZ-61266, Czech Republic, vojnar@fit.vutbr.cz.

2475-1421/2018/1-ART4
https://doi.org/10.1145/3158092

Additional Key Words and Phrases: String Solving, Alternating Finite Automata, Decision Procedure, IC3

ACM Reference Format:
Lukáš Holík, Petr Janků, Anthony W. Lin, Philipp Rümmer, and Tomáš Vojnar. 2018. String Constraints with Concatenation and Transducers Solved Efficiently . *Proc. ACM Program. Lang.* 2, POPL, Article 4 (January 2018), 32 pages. https://doi.org/10.1145/3158092

1 INTRODUCTION

Strings are a fundamental data type in many programming languages. This statement is true now more than ever, especially owing to the rapidly growing popularity of scripting languages (e.g. JavaScript, Python, PHP, and Ruby) wherein programmers tend to make heavy use of string variables. String manipulations are often difficult to reason about automatically, and could easily lead to unexpected programming errors. In some applications, some of these errors could have serious security consequences, e.g., cross-site scripting (a.k.a. XSS), which are ranked among the top three classes of web application security vulnerabilities by OWASP [OWASP 2013].

Popular methods for analysing how strings are being manipulated by a program include *symbolic executions* [Bjørner et al. 2009; Cadar et al. 2008, 2011; Godefroid et al. 2005; Kausler and Sherman 2014; Loring et al. 2017; Redelinghuys et al. 2012; Saxena et al. 2010; Sen et al. 2013] which at their core use constraint solvers over the string domain (a.k.a. *string solvers*). String solvers have been the subject of numerous papers in the past decade, e.g., see [Abdulla et al. 2014; Balzarotti et al. 2008; Barrett et al. 2016; Bjørner et al. 2009; D'Antoni and Veanes 2013; Fu and Li 2010; Fu et al. 2013; Ganesh et al. 2013; Hooimeijer et al. 2011; Hooimeijer and Weimer 2012; Kiezun et al. 2012; Liang et al. 2014, 2016, 2015; Lin and Barceló 2016; Saxena et al. 2010; Trinh et al. 2014, 2016; Veanes et al. 2012; Wassermann et al. 2008; Yu et al. 2010, 2014, 2009, 2011; Zheng et al. 2013] among many others. As is common in constraint solving, we follow the standard approach of *Satisfiability Modulo Theories (SMT)* [De Moura and Bjørner 2011], which is an extension of the problem of satisfiability of Boolean formulae wherein each atomic proposition can be interpreted over some logical theories (typically, quantifier-free).

Unlike the case of constraints over integer/real arithmetic (where many decidability and undecidability results are known and powerful algorithms are already available, e.g., the simplex algorithm), string constraints are much less understood. This is because there are many different string operations that can be included in a theory of strings, e.g., concatenation, length comparisons, regular constraints (matching against a regular expression), and replace-all (i.e. replacing every occurrence of a string by another string). Even for the theory of strings with the concatenation operation alone, existing string solver cannot handle the theory (in its full generality) in a sound and complete manner, despite the existence of a theoretical decision procedure for the problem [Diekert 2002; Gutiérrez 1998; Jez 2016; Makanin 1977; Plandowski 2004, 2006]. This situation is exacerbated when we add extra operations like string-length comparisons, in which case even decidability is a long-standing open problem [Ganesh et al. 2013]. In addition, recent works in string solving have argued in favour of adding the replace-all operator or, more generally finite-state transducers, to string solvers [Lin and Barceló 2016; Trinh et al. 2016; Yu et al. 2010, 2014] in view of their importance for modelling relevant sanitisers (e.g. backslash-escape) and implicit browser transductions (e.g. an application of HTML-unescape by innerHTML), e.g., see [D'Antoni and Veanes 2013; Hooimeijer et al. 2011; Veanes et al. 2012] and Example 1.1 below. However, naively combining the replace-all operator and concatenation yields undecidability [Lin and Barceló 2016].

Example 1.1. The following JavaScript snippet—an adaptation of an example from [Kern 2014; Lin and Barceló 2016]—shows use of *both* concatenation and finite-state transducers:

```
var x = goog.string.htmlEscape(name);
var y = goog.string.escapeString(x);
nameElem.innerHTML = '<button onclick= "viewPerson(\'' + y + '\')">' + x + '</button>';
```

The code assigns an HTML markup for a button to the DOM element `nameElem`. Upon click, the button will invoke the function `viewPerson` on the input `name` whose value is an untrusted variable. The code attempts to first sanitise the value of `name`. This is done via The Closure Library [co 2015] string functions `htmlEscape` and `escapeString`. Inputting the value `Tom & Jerry` into `name` gives the desired HTML markup:

```
<button onclick="viewPerson('Tom &amp;amp; Jerry')">Tom &amp;amp; Jerry</button>
```

On the other hand, inputting value `');attackScript();//` to `name`, results in the markup:

```
<button onclick="viewPerson('&amp;#39;);attackScript();//')">&amp;#39;);attackScript();//')</button>
```

Before this string is inserted into the DOM via `innerHTML`, an implicit browser transduction will take place [Heiderich et al. 2013; Weinberger et al. 2011], i.e., HTML-unescaping the string inside the `onclick` attribute and then invoking the attacker's script `attackScript()` after `viewPerson`. This subtle DOM-based XSS bug is due to calling the right escape functions, but in wrong order. □

One theoretically sound approach proposed in [Lin and Barceló 2016] for overcoming the undecidability of string constraints with both concatenation and finite-state transducers is to impose a *straight-line restriction* on the shape of constraints. This straight-line fragment can be construed as the problem of *path feasibility* [Bjørner et al. 2009] in the following simple imperative language (with only assignment, skip, and assert) for defining non-branching and non-looping string-manipulating programs that are generated by symbolic execution:

$$S ::= y := a \mid \textbf{assert}(b) \mid \textbf{skip} \mid S_1; S_2, \qquad a ::= f(x_1, \ldots, x_n), \qquad b ::= g(x_1)$$

where $f : (\Sigma^*)^n \to \Sigma^*$ is either an application of concatenation $x_1 \circ \cdots \circ x_n$ or an application of a finite-state transduction $R(x_1)$, and g tests membership of x_1 in a regular language. Here, some variables are undefined "input variables". Path feasibility asks if there exist input strings that satisfy all assertions and applications of transductions in the program. It was shown in [Lin and Barceló 2016] that such a path feasibility problem (equivalently, satisfiability for the aforementioned straight-line fragment) is decidable. As noted in [Lin and Barceló 2016] such a fragment can express the program logic of many interesting examples of string-manipulating programs with/without XSS vulnerabilities. For instance, the above example can be modelled as a straight-line formula where the regular constraint comes from an attack pattern like the one below:

```
e1 = /<button onclick=
        "viewPerson\(' ( ' | [^']*[^'\\] ' ) \); [^']*[^'\\]' \)">.*<\/button>/
```

Unfortunately, the decidability proof given in [Lin and Barceló 2016] provides only a theoretical argument for decidability and complexity upper bounds (an exponential-time reduction to the *acyclic fragment* of intersection of rational relations[1] whose decidability proof in turn is a highly intricate polynomial-space procedure using Savitch's trick [Barceló et al. 2013]) and does not yield an implementable solution. Furthermore, despite its decidability, the string logic has a prohibitively high complexity (EXPSPACE-complete, i.e., exponentially higher than without transducers), which could severely limit its applicability.

[1]This fragment consists of constraints that are given as conjunctions of transducers $\bigwedge_{i=1}^{m} R_i(x_i, y_i)$, wherein the graph G of variables does not contain a cycle. The graph G contains vertices corresponding to variables x_i, y_i and that two variables x, y are linked by an edge if $x = x_i$ and $y = y_i$ for some $i \in \{1, \ldots, m\}$.

Contributions. Our paper makes the following contributions to overcome the above challenges:

(1) We propose a fast reduction of satisfiability of formulae in the straight-line fragment and in the acyclic fragment to the emptiness problem of *alternating finite-state automata (AFAs)*. The reduction is in the worst case exponential in the number of concatenation operations[2], but otherwise polynomial in the size of a formula. In combination with fast model checking algorithms (e.g. IC3 [Bradley 2012]) to decide AFA emptiness, this yields the first practical algorithm for handling string constraints with concatenation, finite-state transducers (hence, also replace-all), and regular constraints, and a decision procedure for formulae within the straight-line and acyclic fragments.
(2) We obtain a substantially simpler proof for the decidability and PSPACE-membership of the acyclic fragment of intersection of rational relations of [Barceló et al. 2013], which was crucially used in [Lin and Barceló 2016] as a blackbox in their decidability proof of the straight-line fragment.
(3) We define optimised translations from AFA emptiness to reachability over Boolean transition systems (i.e. which are succinctly represented by Boolean formulae). We implemented our algorithm for string constraints in a new string solver called Sloth, and provide an extensive experimental evaluation. Sloth is the first solver that can handle string constraints that arise from HTML5 applications with sanitisation and implicit browser transductions. Our experiments suggest that the translation to AFAs can circumvent the EXPSPACE worst-case complexity of the straight-line fragment in many practical cases.

An overview of the results. The main technical contribution of our paper is a new method for exploiting alternating automata (AFA) as a succinct symbolic representation for representing formulae in a complex string logic admitting concatenation and finite-state transductions. In particular, the satisfiability problem for the string logic is reduced to AFA language emptiness, for which we exploit fast model checking algorithms. Compared to previous methods [Abdulla et al. 2014; Lin and Barceló 2016] that are based on nondeterministic automata (NFA) and transducers, we show that AFA can incur *at most a linear blowup* for each string operation permitted in the logic (i.e. concatenation, transducers, and regular constraints). While the product NFA representing the intersection of the languages of two automata A_1 and A_2 would be of size $O(|A_1| \times |A_2|)$, the language can be represented using an AFA of size $|A_1| + |A_2|$ (e.g. see [Vardi 1995]). The difficult cases are how to deal with concatenation and replace-all, which are our contributions to the paper. More precisely, a constraint of the form $x := y.z \wedge x \in L$ (where L is the language accepted by an automaton A) was reduced in [Abdulla et al. 2014; Lin and Barceló 2016] to regular constraints on y and z by means of splitting A, which causes a cubic blow-up (since an "intermediate state" in A has to be guessed, and for each state a product of two automata has to be constructed). Similarly, taking the post-image $R(L)$ of L under a relation R represented by a finite-state transducer T gives us an automaton of size $O(|T| \times |A|)$. A naïve application of AFAs is not helpful for those cases, since also projections on AFAs are computationally hard.

The key idea to overcome these difficulties is to *avoid* applying projections altogether, and instead use the AFA to represent general k-ary *rational relations* (a.k.a. k-track finite-state transductions [Barceló et al. 2013; Berstel 1979; Sakarovitch 2009]). This is possible because we focus on formulae without negation, so that the (implicit) existential quantifications for applications of transducers can be placed outside the constraint. This means that our AFAs operate on alphabets that are exponential in size (for k-ary relations, the alphabet is $\{\epsilon, 0, 1\}^k$). To address this problem, we introduce a succinct flavour of AFA with symbolically represented transitions. Our definition is

[2]This is an unavoidable computational limit imposed by EXPSPACE-hardness of the problem [Lin and Barceló 2016].

similar to the concept of alternating symbolic automata in [D'Antoni et al. 2016] with one difference. While symbolic AFA take a transition $q \to_\psi \varphi$ from a state q to a set of states satisfying a formula φ if the input symbol satisfies a formula ψ, our succinct AFA can mix constraints on successor states with those on input symbols within a single transition formula (similarly to the symbolic transition representation of deterministic automata in MONA [Klarlund et al. 2002], where sets of transitions are represented as multi-terminal BDDs with states as terminal nodes). We show how automata splitting can be achieved with at most linear blow-up.

The succinctness of our AFA representation of string formulae is not for free since AFA language emptiness is a PSPACE-complete problem (in contrast to polynomial-time for NFA). However, modern model checking algorithms and heuristics can be harnessed to solve the emptiness problem. In particular, we use a linear-time reduction to reachability in Boolean transition systems similar to [Cox and Leasure 2017; Wang et al. 2016], which can be solved by state of the art model checking algorithms, such as IC3 [Bradley 2012], k-induction [Sheeran et al. 2000], or Craig interpolation-based methods [McMillan 2003], and tools like nuXmv [Cavada et al. 2014] or ABC [Brayton and Mishchenko 2010].

An interesting by-product of our approach is an efficient decision procedure for the acyclic fragment. The acyclic logic does not a priori allow concatenation, but is more liberal in the use of transducer constraints (which can encode complex relations like string-length comparisons, and the subsequence relation). In addition, such a logic is of interest in the investigation of complex path-queries for graph databases [Barceló et al. 2013; Barceló et al. 2012], which has been pursued independently of strings for verification. Our algorithm also yields an alternative and substantially simpler proof of PSPACE upper bound of the satisfiability problem of the logic.

We have implemented our AFA-based string solver as the tool Sloth, using the infrastructure provided by the SMT solver Princess [Rümmer 2008], and applying the nuXmv [Cavada et al. 2014] and ABC [Brayton and Mishchenko 2010] model checkers to analyse succinct AFAs. Sloth is a decision procedure for the discussed fragments of straight-line and acyclic string formulae, and is able to process SMT-LIB input with CVC4-style string operations, augmented with operations `str.replace`, `str.replaceall`[3], and arbitrary transducers defined using sets of mutually recursive functions. Sloth is therefore extremely flexible at supporting intricate string operations, including escape operations such as the ones discussed in Example 1.1. Experiments with string benchmarks drawn from the literature, including problems with replace, replace-all, and general transducers, show that Sloth can solve problems that are beyond the scope of existing solvers, while it is competitive with other solvers on problems with a simpler set of operations.

Organisation. We recall relevant notions from logic and automata theory in Section 2. In Section 3, we define a general string constraint language and mention several important decidable restrictions. In Section 4, we recall the notion of alternating finite-state automata and define a succinct variant that plays a crucial role in our decision procedure. In Section 5, we provide a new algorithm for solving the acyclic fragment of the intersection of rational relations using AFA. In Section 7, we provide our efficient reduction from the straight-line fragment to the acyclic fragment that exploits AFA constructions. To simplify the presentation of this reduction, we first introduce in Section 6 a syntactic sugar of the acyclic fragment called acyclic constraints with synchronisation parameters. In Section 8, we provide our reduction from the AFT emptiness to reachability in a Boolean transition system. Experimental results are presented in Section 9. Our tool Sloth can be obtained from https://github.com/uuverifiers/sloth/wiki. Finally, we conclude in Section 10. Missing proofs can be found in the full version.

[3] `str.replaceall` is the SMT-LIB syntax for the replace-all operation. On the other hand, `str.replace` represents the operation of replacing the *first* occurrence of the given pattern. In case there is no such occurrence, the string stays intact.

2 PRELIMINARIES

Logic. Let $\mathbb{B} = \{0, 1\}$ be the set of Boolean values, and A a set of Boolean variables. We write $\mathbb{F}_A$ to denote the set of *Boolean formulae* over A. In this context, we will sometimes treat subsets A' of A as the corresponding truth assignments $\{s \mapsto 1 \mid s \in A'\} \cup \{s \mapsto 0 \mid s \in A \setminus A'\}$ and write, for instance, $A' \models \varphi$ for $\varphi \in \mathbb{F}_A$ if the assignment satisfies φ. An *atom* is a Boolean variable; a *literal* is either a atom or its negation. A formula is in *disjunctive normal form* (DNF) if it is a disjunction of conjunctions of literals, and in *negation normal form* (NNF) if negation only occurs in front of atoms. We denote the set of variables in a formula φ by $\mathrm{var}(\varphi)$. We use $\bar{x}$ to denote sequences $x_1, \ldots, x_n$ of length $|\bar{x}| = n$ of propositional variables, and we write $\varphi(\bar{x})$ to denote that $\bar{x}$ are the variables of φ. If we do not fix the order of the variables, we write $\varphi(X)$ for a formula with X being its set of variables. For a variable vector $\bar{x}$, we denote by $\{\bar{x}\}$ the set of variables in the vector.

We say that φ is *positive* (*negative*) on an atom $\alpha \in A$ if α appears under an even (odd) number of negations only. A formula that is positive (negative) on all its atoms is called positive (negative), respectively. The constant formulae `true` and `false` are both positive and negative. We use $\mathbb{F}_S^+$ and $\mathbb{F}_S^-$ to denote the sets of all positive and negative Boolean formulae over S, respectively.

Given a formula φ, we write $\widetilde{\varphi}$ to denote a formula obtained by replacing (1) every conjunction by a disjunction and vice versa and (2) every occurrence of `true` by `false` and vice versa. Note that $\widetilde{x} = x$, which means that $\widetilde{\varphi}$ is not the same as the negation of φ.

Strings and languages. Fix a finite alphabet Σ. Elements in Σ^* are interchangeably called words or strings, where the empty word is denoted by ϵ. The concatenation of strings u, v is denoted by $u \circ v$, occasionally just by uv to avoid notational clutter. We denote by $|w|$ the lenght of a word $w \in \Sigma^*$. For any word $w = a_1 \ldots a_n$, $n \geq 1$, and any index $1 \leq i \leq n$, we denote by $w[i]$ the letter a_i. A language is a subset of Σ^*. The concatenation of two languages L, L' is the language $L \circ L' = \{w \circ w' \mid w \in L \wedge w' \in L'\}$, and the iteration L^* of a language L is the smallest language closed under $\circ$ and containing L and ϵ.

Regular languages and rational relations. A regular language over a finite alphabet Σ is a subset of Σ^* that can be built by a finite number of applications of the operations of concatenation, iteration, and union from the languages $\{\epsilon\}$ and $\{a\}, a \in \Sigma$. An n-ary rational relation R over Σ is a subset of $(\Sigma^*)^n$ that can be obtained from a regular language L over the alphabet of n-tuples $(\Sigma \cup \{\epsilon\})^n$ as follows. Include $(w_1, \ldots, w_n)$ in R iff for some $(a_1^1, \ldots, a_n^1), \ldots, (a_1^k, \ldots, a_n^k) \in L$, $w_i = a_1 \circ \cdots \circ a_k$ for all $1 \leq i \leq n$. Here, $\circ$ is a concatenation over the alphabet Σ, and k denotes the length of the words w_i. In practice, regular languages and rational relations can be represented using various flavours of finite-state automata, which are discussed in detail in Section 4.

3 STRING CONSTRAINTS

We start by recalling a general string constraint language from [Lin and Barceló 2016] that supports concatenations, finite-state transducers, and regular expression matching. We will subsequently state decidable fragments of the language for which we design our decision procedure.

3.1 String Language

We assume a vocabulary of countably many *string variables* $x, y, z, \ldots$ ranging over Σ^*. A *string formula* over Σ is a Boolean combination φ of *word equations* $x = t$ whose right-hand side t might contain the concatenation operator, *regular constraints* $P(x)$, and *rational constraints* $\mathcal{R}(\bar{x})$:

$$\varphi \ ::= \ x = t \mid P(x) \mid \mathcal{R}(\bar{x}) \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \neg\varphi, \qquad t \ ::= \ x \mid a \mid t \circ t \ .$$

In the grammar, x ranges over string variables, $\bar{x}$ over vectors of string variables, and $a \in \Sigma$ over letters. $R \subseteq (\Sigma^*)^n$ is assumed to be an n-ary rational relation on words of Σ^*, and $P \subseteq \Sigma^*$ is a regular

language. We will represent regular languages and rational relations by succinct automata and transducers denoted as $\mathcal{R}$ and $\mathcal{A}$, respectively. The automata and transducers will be formalized in Section 4. When the transducer $\mathcal{R}$ or automaton $\mathcal{A}$ representing a rational relation R or regular language P is known, we write $\mathcal{R}(\bar{x})$ or $\mathcal{A}(\bar{x})$ instead of $R(\bar{x})$ or $P(\bar{x})$ in the formulae, respectively.

A formula φ is interpreted over an *assignment* $\iota : \mathrm{var}(\varphi) \to \Sigma^*$ of its variables to strings over Σ^*. It *satisfies* φ, written $\iota \models \varphi$, iff the constraint φ becomes true under the substitution of each variable x by $\iota(x)$. We formalise the satisfaction relation for word equations, rational constraints, and regular constraints, assuming the standard meaning of Boolean connectives:

(1) ι satisfies the equation $x = t$ if $\iota(x) = \iota(t)$, extending ι to terms by setting $\iota(a) = a$ and $\iota(t_1 \circ t_2) = \iota(t_1) \circ \iota(t_2)$.
(2) ι satisfies the rational constraint $\mathcal{R}(x_1, \ldots, x_n)$ iff $(\iota(x_1), \ldots, \iota(x_n))$ belongs to $\mathcal{R}$.
(3) ι satisfies the regular constraint $P(x)$, for P a regular language, if and only if $\iota(x) \in P$.

A satisfying assignment for φ is also called a *solution* for φ. If φ has a solution, it is *satisfiable*.

The unrestricted string logic is undecidable, e.g., one can easily encode Post Correspondence Problem (PCP) as the problem of checking satisfiability of the constraint $\mathcal{R}(x, x)$, for some rational transducer $\mathcal{R}$ [Morvan 2000]. We therefore concentrate on practical decidable fragments.

3.2 Decidable Fragments

Our approach to deciding string formulae is based on two major insights. The first insight is that alternating automata can be used to efficiently decide positive Boolean combinations of rational constraints. This yields an algorithm for deciding (an extension of) the *acyclic fragment* of [Barceló et al. 2013]. The minimalistic definition of acyclic logic restricts rational constraints and does not allow word equations (in Section 5.1 a limited form of equations and arithmetic constraints over lengths will be shown to be encodable in the logic). Our definition of the acyclic logic AC below generalises that of [Barceló et al. 2013] by allowing k-ary rational constraints instead of *binary*.

Definition 3.1 (Acyclic formulae). Particularly, we say that a string formula φ is *acyclic* if it does not contain word equations, rational constraints $\mathcal{R}(x_1, \ldots, x_n)$ only appear positively and their variables $x_1, \ldots, x_n$ are pairwise distinct, and for every sub-formula $\psi \wedge \psi'$ at a positive position of φ (and also every $\psi \vee \psi'$ at a negative position) it is the case that $|\mathrm{free}(\psi) \cap \mathrm{free}(\psi')| \leq 1$, i.e., ψ and ψ' have *at most one* variable in common. We denote by AC the set of all acyclic formulae.

The second main insight we build on is that alternation allows a very efficient encoding of concatenation into rational constraints and automata (though only equisatisfiable, not equivalent). Efficient reasoning about concatenation combined with rational relations is the main selling point of our work from the practical perspective—this is what is most needed and was so far missing in applications like security analysis of web-applications. We follow the approach from [Lin and Barceló 2016] which defines so called straight-line conjunctions. Straight-line conjunctions essentially correspond to sequences of program assignments in the single static assignment form, possibly interleaved with assertions of regular properties. An equation $x = y_1 \circ \cdots \circ y_n$ is understood as an assignment to a program variable x. A rational constraint $\mathcal{R}(x, y)$ may be interpreted as an assignment to x as well, in which case we write it as $x = \mathcal{R}(y)$ (though despite the notation, $\mathcal{R}$ is not required to represent a function, it can still mean any rational relation).

Definition 3.2 (Straight-line conjunction). A conjunction of string constraints is then defined to be *straight-line* if it can be written as $\psi \wedge \bigwedge_{i=1}^{m} x_i = P_i$ where ψ is a conjunction of regular and negated regular constraints and each P_i is either of the form $y_1 \circ \cdots \circ y_n$, or $R(y)$ and, importantly, P_i cannot contain variables $x_i, \ldots, x_m$. We denote by SL the set of all straight-line conjunctions.

Example 3.3. The program snippet in Example 1.1 would be expressed as $x = \mathcal{R}_1(\text{name}) \wedge y = \mathcal{R}_2(x) \wedge z = w_1 \circ y \circ w_2 \circ x \circ w_3 \wedge u = \mathcal{R}_3(z)$. The transducers $\mathcal{R}_i$ correspond to the string operations at the respective lines: $\mathcal{R}_1$ is the htmlEscape, $\mathcal{R}_2$ is the escapeString, and $\mathcal{R}_3$ is the implicit transduction within innerHTML. Line 3 is translated into a conjunction of the concatenation and the third rational constraint encoding the implicit string operation at the assignment to innerHTML. In the concatenation, w_1, w_2, w_3 are words that correspond to the three constant strings concatenated with x and y on line 3. To test vulnerability, a regular constraint $\mathcal{A}(u)$ encoding the pattern e1 is added as a conjunct.

The fragment of straight-line conjunctions can be straightforwardly extended to disjunctive formulae. We say that a string formula is straight-line if every clause in its DNF is straight-line. A decision procedure for straight-line conjunctions immediately extends to straight-line formulae: instantiate the DPLL(T) framework [Nieuwenhuis et al. 2004] with a solver for straight-line conjunctions.

The straight-line and acyclic fragments are clearly syntactically incomparable: AC does not have equations, SL restricts more strictly combinations of rational relations and allows only binary ones. Regarding expressive power, SL can express properties which AC cannot: the straight-line constraint $x = yy$ cannot be expressed by any acyclic formula. On the other hand, whether or not AC formulae can be expressed in SL is not clear. Every AC formula can be expressed by a single n-ary acyclic rational constraint (c.f. Section 5), hence acyclic formulae and acyclic rational constraints are of the same power. It is not clear however whether straight-line formulae, which can use only binary rational constraints, can express arbitrary n-ary acyclic rational constraint.

4 SUCCINCT ALTERNATING AUTOMATA AND TRANSDUCERS

We introduce a succinct form of alternating automata and transducers that operate over *bit vectors*, i.e., functions $b : V \to \mathbb{B}$ where V is a finite, totally ordered set of bit variables. This is a variant of the recent automata model in [D'Antoni et al. 2016] that is tailored to our problem. Bit vectors can of course be described by strings over $\mathbb{B}$, conjunctions of literals over V, or sets of those elements $v \in V$ such that $b(v) = 1$. In what follows, we will use all of these representations interchangeably. Referring to the last mentioned possibility, we denote the set of all bit vectors over V by $\mathcal{P}(V)$.

An obvious advantage of this approach is that encoding symbols of large alphabets, such as UTF, by bit vectors allows one to succinctly represent sets of such symbols using Boolean formulae. In particular, symbols of an alphabet of size 2^k can be encoded by bit vectors of size k (or, alternatively, as Boolean formulae over k Boolean variables). We use this fact when encoding transitions of our alternating automata.

Example 4.1. To illustrate the encoding, assume the alphabet $\Sigma = \{a, b, c, d\}$ consisting of symbols a, b, c, and d. We can deal with this alphabet by using the set $V = \{v_0, v_1\}$ and representing, e.g., a as $\neg v_1 \wedge \neg v_0$, b as $\neg v_1 \wedge v_0$, c as $v_1 \wedge \neg v_0$, and d as $v_1 \wedge v_0$. This is, a, b, c, and d are encoded as the bit vectors 00, 01, 10, and 11 (for the ordering $v_0 < v_1$), or the sets $\emptyset$, $\{v_0\}$, $\{v_1\}$, $\{v_0, v_1\}$, respectively. The set of symbols $\{c, d\}$ can then be encoded simply by the formula v_1. □

4.1 Succinct Alternating Finite Automata

A *succinct alternating finite automaton (AFA)* over Boolean variables V is a tuple $\mathcal{A} = (V, Q, \Delta, I, F)$ where Q is a finite set of *states*, the *transition function* $\Delta : Q \to \mathbb{F}_{V \cup Q}$ assigns to every state a Boolean formula over Boolean variables and states that is positive on states, $I \in \mathbb{F}^+_Q$ is a positive *initial formula*, and $F \in \mathbb{F}^-_Q$ is a negative *final formula*. Let $w = b_1 \ldots b_m$, $m \geq 0$, be a word where each b_i, $1 \leq i \leq m$, is a bit vector encoding the i-th letter of w. A *run* of the AFA $\mathcal{A}$ over w is a sequence $\rho = \rho_0 b_1 \rho_1 \ldots b_m \rho_m$ where $b_i \in \mathcal{P}(V)$ for every $1 \leq i \leq m$, $\rho_i \subseteq Q$ for every $0 \leq i \leq m$, and

$b_i \cup \rho_i \models \bigwedge_{q \in \rho_{i-1}} \Delta(q)$ for every $1 \leq i \leq m$. The run is *accepting* if $\rho_0 \models I$ and $\rho_m \models F$, in which case the word is accepted. The *language* of $\mathcal{A}$ is the set $L(\mathcal{A})$ of accepted words.

Notice that instead of the more usual definition of Δ, which would assign a positive Boolean formula over Q to every pair from $Q \times \mathcal{P}(V)$ or to a pair $Q \times \mathbb{F}_V$ as in [D'Antoni et al. 2016], we let Δ assign to states formulae that talk about both target states and Boolean input variables. This is closer to the encoding of the transition function as used in MONA [Klarlund et al. 2002]. It allows for additional succinctness and also for a more natural translation of the language emptiness problem into a model checking problem (cf. Section 8).[4] Moreover, compared with the usual AFA definition, we do not have just a single initial state and a single set of accepting states, but we use initial and final formulae. As will become clear in Section 5, this approach allows us to easily translate the considered formulae into AFAs in an inductive way.

Note that standard *nondeterministic finite automata* (NFAs), working over bit vectors, can be obtained as a special case of our AFAs as follows. An AFA $\mathcal{A} = (V, Q, \Delta, I, F)$ is an NFA iff (1) I is of the form $\bigvee_{q \in Q'} q$ for some $Q' \subseteq Q$, (2) F is of the form $\bigwedge_{q \in Q''} \neg q$ for some $Q'' \subseteq Q$, and (3) for every $q \in Q$, $\Delta(q)$ is of the form $\bigvee_{1 \leq i \leq m} \varphi_i(V) \wedge q_i$ where $m \geq 0$ and, for all $1 \leq i \leq m$, $\varphi_i(V)$ is a formula over the input bit variables and $q_i \in Q$.

Example 4.2. To illustrate our notion of AFAs, we give an example of an AFA $\mathcal{A}$ over the alphabet $\Sigma = \{a, b, c, d\}$ from Example 4.1 that accepts the language $\{w \in \Sigma^* \mid |w| \bmod 35 = 0 \wedge \forall i \exists j : (1 \leq i \leq |w| \wedge w[i] \in \{a, b\}) \rightarrow (i < j \leq |w| \wedge w[j] \in \{c, d\})\}$, i.e., the length of the words is a multiple of 35, and every letter a or b is eventually followed by a letter c or d. In particular, we let $\mathcal{A} = (\{v_0, v_1\}, \{q_0, \ldots, q_4, p_0, \ldots, p_6, r_1, r_2\}, \Delta, I, F)$ where $I = q_0 \wedge p_0$, $F = \neg q_1 \wedge \ldots \wedge \neg q_4 \wedge \neg p_1 \wedge \ldots \wedge \neg p_6 \wedge \neg r_1$ (i.e., the accepting states are q_0, p_0, and r_2), and Δ is defined as follows:

- $\forall 0 \leq i < 5 : \Delta(q_i) = (\neg v_1 \wedge q_{(i+1) \bmod 5} \wedge r_1) \vee (v_1 \wedge q_{(i+1) \bmod 5})$,
- $\forall 0 \leq i < 7 : \Delta(p_i) = p_{(i+1) \bmod 7}$,
- $\Delta(r_1) = (v_1 \wedge r_2) \vee (\neg v_1 \wedge r_1)$ and $\Delta(r_2) = r_2$.

Intuitively, the q states check divisibility by 5. Moreover, whenever, they encounter an a or b symbol (encoded succinctly as checking $\neg v_1$ in the AFA), they spawn a run through the r states, which checks that eventually a c or d symbol appears. The p states then check divisibility by 7. The desired language is accepted due to the requirement that all these runs must be synchronized. Note that encoding the language using an NFA would require quadratically more states since an explicit product of all the branches would have to be done. □

The additional succinctness of AFA does not influence the computational complexity of the emptiness check compared to the standard variant of alternating automata.

Lemma 4.3. *The problem of language emptiness of AFA is PSPACE-complete.*

The lemma is witnessed by a linear-space transformation of the problem of emptiness of an AFA language to the PSPACE-complete problem of reachability in a Boolean transition system. This transformation is shown in Section 8.

4.2 Boolean Operations on AFAs

From the standard Boolean operations over AFAs, we will mainly need conjunction and disjunction in this paper. These operations can be implemented in linear space and time in a way analogous to [D'Antoni et al. 2016], slightly adapted for our notion of initial/final formulae, as follows. Given

[4] [D'Antoni et al. 2016] also mentions an implementation of symbolic AFAs that uses MONA-like BDDs and is technically close to our AFAs.

two AFAs $\mathcal{A} = (V, Q, \Delta, I, F)$ and $\mathcal{A}' = (V, Q', \Delta', I', F')$ with $Q \cap Q' = \emptyset$, the automaton accepting the union of their languages can be constructed as $\mathcal{A} \cup \mathcal{A}' = (V, Q \cup Q', \Delta \cup \Delta', I \vee I', F \wedge F')$, and the automaton accepting the intersection of their languages can be constructed as $\mathcal{A} \cap \mathcal{A}' = (V, Q \cup Q', \Delta \cup \Delta', I \wedge I', F \wedge F')$. Seeing correctness of the construction of $\mathcal{A} \cap \mathcal{A}'$ is immediate. Indeed, the initial condition enforces that the two AFAs run in parallel, disjointness of their state-spaces prevents them from influencing one another, and the final condition defines their parallel runs as accepting iff both of the runs accept. To see correctness of the construction of $\mathcal{A} \cup \mathcal{A}'$, it is enough to consider that one of the automata can be started with the empty set of states (corresponding to the formula $\bigwedge_{q \in Q} \neg q$ for $\mathcal{A}$ and likewise for $\mathcal{A}'$). This is possible since only one of the initial formulae I and I' needs to be satisfied. The automaton that was started with the empty set of states will stay with the empty set of states throughout the entire run and thus trivially satisfy the (negative) final formula.

Example 4.4. Note that the AFA in Example 4.2 can be viewed as obtained by conjunction of two AFAs: one consisting of the q and r states and the second of the p states. □

To complement an AFA $\mathcal{A} = (V, Q, \Delta, I, F)$, we first transform the automaton into a form corresponding to the symbolic AFA of [D'Antoni et al. 2016] and then use their complementation procedure. More precisely, the transformation to the symbolic AFA form requires two steps:

- The first step simplifies the final condition. The final formula F is converted into DNF, yielding a formula $F_1 \vee \ldots \vee F_k$, $k \geq 1$, where each F_i, $1 \leq i \leq k$, is a conjunction of negative literals over Q. The AFA $\mathcal{A}$ is then transformed into a union of AFAs $\mathcal{A}_i = (V, Q, \Delta, I, F_i)$, $1 \leq i \leq k$, where each $\mathcal{A}_i$ is a copy of $\mathcal{A}$ except that it uses one of the disjuncts F_i of the DNF form of the original final formula F as its final formula. Each resulting AFAs hence have a purely conjunctive final condition that corresponds a set of final states of [D'Antoni et al. 2016] (a set of final states $F \subseteq Q$ would correspond to the final formula $\bigwedge_{q \in Q \setminus F} \neg q$).
- The second step simplifies the structure of the transitions. For every $q \in Q$, the transition formula $\Delta(q)$ is transformed into a disjunction of formulae of the form $(\varphi_1(V) \wedge \psi_1(Q)) \vee \ldots \vee (\varphi_m(V) \wedge \psi_m(Q))$ where the $\varphi_i(V)$ formulae, called *input formulae* below, speak about input bit variables only, while the $\psi_i(Q)$ formulae, called *target formulae* below, speak exclusively about the target states, for $1 \leq i \leq m$. For this transformation, a slight modification of transforming a formula into DNF can be used.

The complementation procedure of [D'Antoni et al. 2016] then proceeds in two steps: the *normalisation* and the complementation itself. We sketch them below:

- For every $q \in Q$, normalisation transforms the transition formula $\Delta(q) = (\varphi_1(V) \wedge \psi_1(Q)) \vee \ldots \vee (\varphi_m(V) \wedge \psi_m(Q))$ so that every two distinct input formulae $\varphi(V)$ and $\varphi'(V)$ of the resulting formula describe disjoint sets of bit vectors, i.e., $\neg(\varphi(V) \wedge \varphi'(V))$ holds. To achieve this (without trying to optimize the algorithm as in [D'Antoni et al. 2016]), one can consider generating all Boolean combinations of the original $\varphi(V)$ formulae, conjoining each of them with the disjunction of those state formulae whose input formulae are taken positively in the given case. More precisely, one can take $\bigvee_{I \subseteq \{1,\ldots,m\}} (\bigwedge_{i \in I} \varphi_i)) \wedge (\bigwedge_{i \in \{1,\ldots,m\} \setminus I} \neg \varphi_i)) \wedge \bigvee_{i \in I} \psi_i$.
- Finally, to complement the AFAs normalized in the above way, one proceeds as follows: (1) The initial formula I is replaced by $\widetilde{I}$. (2) For every $q \in Q$ and every disjunct $\varphi(V) \wedge \psi(Q)$ of the transition formula $\Delta(q)$, the target formula $\psi(Q)$ is replaced by $\widetilde{\psi}(Q)$. (3) The final formula of the form $\bigwedge_{q \in Q'} \neg q$, $Q' \subseteq Q$, is transformed to the formula $\bigwedge_{q \in Q \setminus Q'} \neg q$, and `false` is swapped for `true` and vice versa.

Clearly, the complementation contains three sources of exponential blow-up: (1) the simplification of the final condition, (2) the simplification of transitions and (3) the normalization of transitions.

Note, however, that, in this paper, we will apply complementation exclusively on AFAs obtained by Boolean operations from NFAs derived from regular expressions. Such AFAs already have the simple final conditions, and so the first source of exponential blow-up does not apply. The second and the third source of exponential complexity can manifest themselves but note that it does not show up in the number of states. Finally, note that if we used AFAs with explicit alphabets, the second and the third problem would disappear (but then the AFAs would usually be bigger anyway).

4.3 Succinct Alternating Finite Transducers

In our alternating finite transducers, we will need to use *epsilon symbols* representing the empty word. Moreover, as we will explain later, in order to avoid some undesirable synchronization when composing the transducers, we will need more such symbols—differing just syntactically. Technically, we will encode the epsilon symbols using a set of epsilon bit variables E, containing one new bit variable for each epsilon symbol. We will draw the epsilon bit variables from a countably infinite set $\mathcal{E}$. We will also assume that when one of these bits is set, other bits are not important.

Let W be a finite, totally ordered set of bit variables, which we can split to the set of input bit variables $V(W) = W \setminus \mathcal{E}$ and the set of epsilon bit variables $E(W) = W \cap \mathcal{E}$. Given a word $w = b_1 \dots b_m \in \mathcal{P}(W)^*$, $m \geq 0$, we denote by $\rangle w \langle$ the word that arises from w by erasing all those b_i, $1 \leq i \leq m$, in which some epsilon bit variable is set, i.e., $b_i \cap \mathcal{E} \neq \emptyset$. Further, let $k \geq 1$, and let $W\langle k \rangle = W \times [k]$, assuming it to be ordered in the lexicographic way. The indexing of the bit variables will be used to express the track on which they are read. Finally, given a word $w = b_1 \dots b_m \in \mathcal{P}(W\langle k \rangle)^*$, $m \geq 0$, we denote by $w\downarrow_i$, $1 \leq i \leq k$, the word $b'_1 \dots b'_m \in \mathcal{P}(W)^*$ that arises from w by keeping the contents of the i-th track (without the index i) only, i.e., $b'_j \times \{i\} = b_j \cap (W \times \{i\})$ for $1 \leq j \leq m$.

A k-track *succinct alternating finite transducer* (AFT) over W is syntactically an alternating automaton $\mathcal{R} = (W\langle k \rangle, Q, \Delta, I, F), k \geq 1$. Let $V = V(W)$. The relation $Rel(\mathcal{R}) \subseteq (\mathcal{P}(V)^*)^k$ *recognised* by $\mathcal{R}$ contains a k-tuple of words $(x_1, \dots, x_k)$ over $\mathcal{P}(V)$ iff there is a word $w \in L(\mathcal{R})$ such that $x_i = \rangle w\downarrow_i \langle$ for each $1 \leq i \leq k$.

Below, we will sometimes say that the word w encodes the k-tuple of words $(x_1, \dots, x_k)$. Moreover, for simplicity, instead of saying that $\mathcal{R}$ has a run over w that encodes $(x_1, \dots, x_k)$, we will sometimes directly say that $\mathcal{R}$ has a run over $(x_1, \dots, x_k)$ or that $\mathcal{R}$ accepts $(x_1, \dots, x_k)$.

Finally, note that classical *nondeterministic finite transducers* (NFTs) are a special case of our AFTs that can be defined by a similar restriction as the one used when restricting AFAs to NFAs. In particular, the first track (with letters indexed with 1) can be seen as the input track, and the second track (with letters indexed with 2) can be seen as the output track. AFTs as well as NFTs recognize the class of *rational relations* [Barceló et al. 2013; Berstel 1979; Sakarovitch 2009].

Example 4.5. We now give a simple example of an AFT that implements escaping of every apostrophe by a backlash in the UTF-8 encoding. Intuitively, the AFT will transform an input string `x'xx` to the string `x\'xx`, i.e., the relation it represents will contain the couple (`x'xx`,`x\'xx`). All the symbols should, however, be encoded in UTF-8. In this encoding, the apostrophe has the binary code 00100111, and the backlash has the code 00101010. We will work with the set of bit variables $V_8 = \{v_0, \dots, v_7\}$ and a single epsilon bit variable e. We will superscript the bit variables by the track on which they are read (hence, e.g., v_1^2 is the same as $(v_1, 2)$, i.e., v_1 is read on the second track). Let $\mathsf{ap}^i = v_0^i \wedge v_1^i \wedge v_2^i \wedge \neg v_3^i \wedge \neg v_4^i \wedge v_5^i \wedge \neg v_6^i \wedge \neg v_7^i \wedge \neg e^i$ represent an apostrophe read on the i-th track. Next, let $\mathsf{bc}^i = \neg v_0^i \wedge v_1^i \wedge \neg v_2^i \wedge v_3^i \wedge \neg v_4^i \wedge v_5^i \wedge \neg v_6^i \wedge \neg v_7^i \wedge \neg e^i$ represent a backlash read on the i-th track. Finally, let $\mathsf{eq}^{i,j} = e^i \leftrightarrow e^j \wedge \bigwedge_{0 \leq k < 8} v_k^i \leftrightarrow v_k^j$ denote that the same symbol is read on the i-th and j-th track. The AFT that implements the described escaping can

be constructed as follows: $\mathcal{R} = ((V_8 \cup \{e\})\langle 2\rangle, \{q_0, q_1\}, \Delta, q_0, \neg q_1)$ where the transition formulae are defined by $\Delta(q_0) = (\neg \mathsf{ap}^1 \wedge \mathsf{eq}^{1,2} \wedge q_0) \vee (\mathsf{ap}^1 \wedge \mathsf{bc}^2 \wedge q_1)$ and $\Delta(q_1) = e^1 \wedge \mathsf{ap}^2 \wedge q_0$. □

5 DECIDING ACYCLIC FORMULAE

Our decision procedure for AC formulae is based on translating them into AFTs. For simplicity, we assume that the formula is negation free (after transforming to NNF, negation at regular constraints can be eliminated by AFA complementation). Notice that with no negations, the restriction AC puts on disjunctions never applies. We also assume that the formula contains rational constraints only (regular constraint can be understood as unary rational constraints).

Our algorithm then transforms a formula $\varphi(\bar{x})$ into a rational constraint $\mathcal{R}_\varphi(\bar{x})$ inductively on the structure of φ. As the base case, we get rational constraints $\mathcal{R}(\bar{x})$, which are already represented as AFTs, and regular constraints $\mathcal{A}(x)$, already represented by AFAs. Boolean operations over regular constraints can be treated using the corresponding Boolean operations over AFAs described in Section 4.2. The resulting AFAs can then be viewed as rational constraints with one variable (and hence as a single-track AFT).

Once constraints $\mathcal{R}_\varphi(\bar{x})$ and $\mathcal{R}_\psi(\bar{y})$ are available, the induction step translates formulae $\mathcal{R}_\varphi(\bar{x}) \wedge \mathcal{R}_\psi(\bar{y})$ and $\mathcal{R}_\varphi(\bar{x}) \vee \mathcal{R}_\psi(\bar{y})$ to constraints $\mathcal{R}_{\varphi\wedge\psi}(\bar{z})$ and $\mathcal{R}_{\varphi\vee\psi}(\bar{z})$, respectively. To be able to describe this step in detail, let $\mathcal{R}_\varphi = ((V \cup E_\varphi)\langle|\bar{x}|\rangle, Q_\varphi, \Delta_\varphi, I_\varphi, F_\varphi)$ and $\mathcal{R}_\psi = ((V \cup E_\psi)\langle|\bar{y}|\rangle, Q_\psi, \Delta_\psi, I_\psi, F_\psi)$ such that w.l.o.g. $Q_\varphi \cap Q_\psi = \emptyset$ and $E_\varphi \cap E_\psi = \emptyset$.

Translation of conjunctions to AFTs. The construction of $\mathcal{R}_{\varphi\wedge\psi}$ has three steps:

(1) **Alignment of tracks** that ensures that distinct variables are assigned different tracks and that the transducers agree on the track used for the shared variable.
(2) **Saturation by ϵ-self loops** allowing the AFTs to synchronize whenever one of them makes an ϵ move on the shared track.
(3) **Conjunction** on the resulting AFTs viewing them as AFAs.

Alignment of tracks. Given constraints $\mathcal{R}_\varphi(\bar{x})$ and $\mathcal{R}_\psi(\bar{y})$, the goal of the alignment of tracks is to assign distinct tracks to distinct variables of $\bar{x}$ and $\bar{y}$, and to assign the same track in both of the transducers to the shared variable—if there is one (recall that, by acyclicity, $\bar{x}$ and $\bar{y}$ do not contain repeating variables and share at most one common variable). This is implemented by choosing a vector $\bar{z}$ that consists of exactly one occurrence of every variable from $\bar{x}$ and $\bar{y}$, i.e., $\{\bar{z}\} = \{\bar{x}\} \cup \{\bar{y}\}$, and by subsequently re-indexing the bit vector variables in the transition relations. Particularly, in Δ_φ, every indexed bit vector variable v^i (including epsilon bit variables) is replaced by v^j with j being the position of x_i in $\bar{z}$, and analogously in Δ_ψ, every indexed bit variable v^i is replaced by v^j with j being the position of y_i in $\bar{z}$. Both AFTs are then considered to have $|\bar{z}|$ tracks.

Saturation by ϵ-self loops. This step is needed if $\bar{x}$ and $\bar{y}$ share a variable, i.e., $\{\bar{x}\} \cap \{\bar{y}\} \neq \emptyset$. The two input transducers then have to synchronise on reading its symbols. However, it may happen that, at some point, one of them will want to read from the non-shared tracks exclusively, performing an ϵ transition on the shared track. Since reading of the non-shared tracks can be ignored by the other transducer, it should be allowed to perform an ϵ move on all of its tracks. However, that needs not be allowed by its transition function. To compensate for this, we will saturate the transition function by ϵ-self loops performed on all tracks. Unfortunately, there is one additional problem with this step: If the added ϵ transitions were based on the same epsilon bit variables as those already used in the given AFT, they could enable some additional synchronization *within* the given AFT, thus allowing it to accept some more tuples of words. We give an example of this problem below (Example 5.2). To resolve the problem, we assume that the two AFTs being conjuncted use different epsilon bit variables (more of such variables can be used due the AFTs can be a result of

several previous conjunctions). Formally, for any choice $\sigma, \sigma' \in \{\varphi, \psi\}$ such that $\sigma \neq \sigma'$, and for every state $q \in Q_\sigma$, the transition formula $\Delta_\sigma(q)$ is replaced by $\Delta_\sigma(q) \vee (q \wedge \bigvee_{e \in E_{\sigma'}} \bigwedge_{i \in [|\bar{z}|]} e^i)$.

Conjunction of AFTs viewed as AFAs. In the last step, the input AFTs with aligned tracks and saturated by ϵ-self loops are conjoined using the automata intersection construction from Section 4.2.

LEMMA 5.1. *Let $\mathcal{R}'_\varphi$ and $\mathcal{R}'_\psi$ be the AFTs obtained from the input AFTs $\mathcal{R}_\varphi$ and $\mathcal{R}_\psi$ by track alignment and ϵ-self-loop saturation, and let $\mathcal{R}_{\varphi\wedge\psi} = \mathcal{R}'_\varphi \cap \mathcal{R}'_\psi$. Then, $\mathcal{R}_{\varphi\wedge\psi}(\bar{z})$ is equivalent to $\mathcal{R}_\varphi(\bar{x}) \wedge \mathcal{R}_\psi(\bar{y})$.*

To see that the lemma holds, note that both $\mathcal{R}'_\varphi$ and $\mathcal{R}'_\psi$ have the same number of tracks—namely, $|\bar{z}|$. This number can be bigger than the original number of tracks ($|\bar{x}|$ or $|\bar{y}|$, resp.), but the AFTs still represent the same relations over the original tracks (the added tracks are unconstrained). The ϵ-self loop saturation does not alter the represented relations either as the added transitions represent empty words across all tracks only, and, moreover, they cannot synchronize with the original transitions, unblocking some originally blocked runs. Finally, due to the saturation, the two AFTs cannot block each other by an epsilon move on the shared track available in one of them only.[5]

Example 5.2. We now provide an example illustrating the conjunction of AFTs, including the need to saturate the AFTs by ϵ-self loops with different ϵ symbols. We will assume working with the input alphabet $\Sigma = \{a, b\}$ encoded using a single input bit variable v_0: let a correspond to $\neg v_0$ and b to v_0. Moreover, we will use two epsilon bit variables, namely, e_1 and e_2. We consider the following two simple AFTs, each with two tracks:

- $\mathcal{R}_1 = (\{v_0, e_1\}\langle 2\rangle, \{q_0, q_1, q_2\}, \Delta_1, q_0, \neg q_0 \wedge \neg q_2)$ with $\Delta_1(q_0) = (a^1 \wedge b^2 \wedge q_1) \vee (a^1 \wedge a^2 \wedge q_1 \wedge q_2)$, $\Delta_1(q_1) = \texttt{false}$, and $\Delta_1(q_2) = e_1^1 \wedge q_1$. Note that $Rel(\mathcal{R}_1) = \{(a, b)\}$ since the run that starts with $a^1 \wedge a^2$ gets stuck in one of its branches, namely the one that goes to q_2. This is because we require branches of a single run of an AFT to synchronize even on epsilon bit variables, and the transition from q_2 cannot synchronize with any move from q_1.
- $\mathcal{R}_2 = (\{v_0, e_2\}\langle 2\rangle, \{p_0, p_1, p_2\}, \Delta_2, p_0, \neg p_0 \wedge \neg p_1)$ such that $\Delta_2(p_0) = (a^1 \wedge b^2 \wedge p_1)$, $\Delta_2(p_1) = e_2^1 \wedge b^2 \wedge p_2$, and $\Delta_2(p_2) = \texttt{false}$. Clearly, $Rel(\mathcal{R}_2) = \{(a, bb)\}$.

Let Q_i, I_i, F_i denote the set of states, initial constraint, and final constraint of $\mathcal{R}_i$, $i \in \{1, 2\}$, respectively. Assume that we want to construct an AFT for the constraint $\mathcal{R}_1(x, y) \wedge \mathcal{R}_2(x, z)$. This constraint represents the ternary relation $\{(a, b, bb)\}$. It can be seen that if we apply the above described construction for intersection of AFTs to $\mathcal{R}'_1$ and $\mathcal{R}'_2$, where $\mathcal{R}'_1 = \mathcal{R}_1$ and $\mathcal{R}'_2$ is the same as $\mathcal{R}_2$ up to all symbols from track to 2 are moved to track 3, we will get an AFT $\mathcal{R} = (\{v_0, e_1, e_2\}\langle 3\rangle, Q_1 \cup Q_2, \Delta, I_1 \wedge I_2, F_1 \wedge F_2)$ representing exactly this relation. We will not list here the entire Δ but let us note the below:

- Δ will contain the following transition obtained by ϵ-self-loop saturation of $\mathcal{R}_1$: $\Delta(q_1) = (e_2^1 \wedge e_2^2 \wedge q_1)$. This will allow $\mathcal{R}$ to synchronize its run through q_1 with its run from p_1 to p_2. Without the saturation, this would not be possible, and $Rel(\mathcal{R})$ would be empty.
- On the other hand, if a single epsilon bit variable e was used in both AFTs as well as in their saturation, the saturated Δ_1 would include the transition $\Delta_1(q_1) = (e^1 \wedge e^2 \wedge q_1)$. This transition could synchronize with the transition $\Delta_1(q_2) = e^1 \wedge q_1$, and the relation represented by the saturated $\mathcal{R}_1$ would grow to $Rel(\mathcal{R}_1) = \{(a, b), (a, a)\}$. The result of the intersection would then (wrongly) represent the relation $\{(a, b, bb), (a, a, bb)\}$. □

[5]Note that the same approach cannot be used for AFTs sharing more than one track. Indeed, by intersecting two general rational relations, one needs not obtain a rational relation.

Translation of disjunctions to AFTs. The construction of an AFT for a disjunction of formulae is slightly simpler. The alignment of variables is immediately followed by an application of the AFA disjunction construction. That is, the AFT $\mathcal{R}_{\varphi\vee\psi}$ is constructed simply as $\mathcal{R}'_\varphi \cup \mathcal{R}'_\psi$ from the constraints $\mathcal{R}'_\varphi(\bar{z})$ and $\mathcal{R}'_\psi(\bar{z})$ produced by the alignment of the vectors of variables $\bar{x}$ and $\bar{y}$ in $\mathcal{R}_\varphi(\bar{x})$ and $\mathcal{R}_\psi(\bar{y})$. The construction of $\mathcal{R}'_\varphi$ and $\mathcal{R}'_\psi$ does not require the saturation by ϵ-self loops because the two transducers do not need to synchronise on reading shared variables. The vectors $\bar{x}$ and $\bar{y}$ are allowed to share any number of variables.

Theorem 5.3. *Every acyclic formula $\varphi(\bar{x})$ can be transformed into an equisatisfiable rational constraint $\mathcal{R}(\bar{x})$ represented by an AFT $\mathcal{R}$. The transformation can be done in polynomial time unless φ contains a negated regular constraint represented by a non-normalized succinct NFA.*

Corollary 5.4. *Checking satisfiability of acyclic formulae is in PSPACE unless the formulae contain a negated regular constraint represented by a non-normalized succinct NFA.*

PSPACE membership of satisfiability of acyclic formulae with binary rational constraints (without negations of regular constraints and without considering succinct alphabet encoding) is proven already in [Barceló et al. 2013]. Apart from extending the result to k-ary rational constraints, we obtain a simpler proof as a corollary of Theorem 5.3, avoiding a need to use the highly intricate polynomial-space procedure based on the Savitch´s trick used in [Barceló et al. 2013]. Not considering the problem of negating regular constraints, our PSPACE algorithm would first construct a linear-size AFT for the input φ. We can then use the fact that the standard PSPACE algorithm for checking emptiness of AFAs/AFTs easily generalises to succinct AFAs/AFTs. This is proved by our linear-space reduction of emptiness of the language of succinct AFAs to reachability in Boolean transition systems, presented in Section 8. Reachability in Boolean transition systems is known to be PSPACE-complete.

5.1 Decidable Extensions of AC

The relatively liberal condition that AC puts on rational constraints allow us to easily extend AC with other features, without having to change the decision procedure. Namely, we can add Presburger constraints about word length, as well as word equations, as long as overall acyclicity of a formula is preserved. Length constraints can be added in the general form $\varphi_{\text{Pres}}(|x_1|, \ldots, |x_k|)$, where φ_{Pres} is a Presburger formula.

Definition 5.5 (Extended acyclic formulae). A string formula φ augmented with length constraints $\varphi_{\text{Pres}}(|x_1|, \ldots, |x_k|)$ is *extended acyclic* if every word equation or rational constraint contains each variable at most once, rational constraints $\mathcal{R}(x_1, \ldots, x_n)$ only appear at positive positions, and for every sub-formula $\psi \wedge \psi'$ at a positive position of φ (and also every $\psi \vee \psi'$ at a negative position) it is the case that $|\text{free}(\psi) \cap \text{free}(\psi')| \leq 1$, i.e., ψ and ψ' have *at most one* variable in common.

Any extended AC formula φ can be turned into a standard AC formula by translating word equations and length constraints to rational constraints. Notice that, although quite powerful, extended AC still cannot express SL formulae such as $x = yy$, and does not cover practical properties such as, e.g., those in Example 3.3 (where two conjuncts contain both x and y).

Word equations to rational constraints. For simplicity, assume that equations do not contain letters $a \in \Sigma$. This can be achieved by replacing every occurrence of a constraint b by a fresh variable constrained by the regular language $\{b\}$. An equation $x = x_1 \circ \cdots \circ x_n$ without multiple occurrences of any variables is translated to a rational constraint $\mathcal{R}(x, x_1, \ldots, x_n)$ with $\mathcal{R} = (W\langle n+1\rangle, Q =$

$\{q_0, \ldots, q_n\}, \Delta, I = q_0, F = q_n$). The transitions for $i \in [n]$ are

$$\Delta(q_{i-1}) = (q_{i-1} \vee q_i) \wedge \bigwedge_{j \in [n] \setminus \{i\}} e^j \wedge \bigwedge_{v \in W\langle n+1 \rangle} (v^i \leftrightarrow v^0).$$

and $\Delta(q_n) = \texttt{false}$. That is, the symbol on the first track is copied to the ith track while all the other tracks read ϵ. Negated word equations can be translated to AFTs in a similar way.

Length constraints to rational constraints. The translation of length constraints to rational constraints is similarly straightforward. Suppose an extended AC formula contains a length constraint $\varphi_{\text{Pres}}(|x_1|, \ldots, |x_k|)$, where φ_{Pres} is a Presburger formula over k variables $y_1, \ldots, y_k$ ranging over natural numbers. It is a classical result that the solution space of φ_{Pres} forms a semi-linear set [Ginsburg and Spanier 1966], i.e., can be represented as a finite union of linear sets $L_j = \{\bar{y}_0 + \sum_{i=1}^{m} \lambda_i \bar{y}_i \mid \lambda_1, \ldots, \lambda_m \in \mathbb{N}\} \subseteq \mathbb{N}^k$ with $\bar{y}_0, \ldots \bar{y}_m \in \mathbb{N}^k$. Every linear set L_j can directly be translated to a succinct k-track AFT recognising the relation $\{(x_1, \ldots, x_k) \in (\Sigma^*)^k \mid (|x_1|, \ldots, |x_k|) \in L_j\}$, and the union of AFTs be constructed as shown in Section 4.2, resulting in an AFT $\mathcal{R}_{\varphi_{\text{Pres}}}(x_1, \ldots, x_k)$ that is equivalent to $\varphi_{\text{Pres}}(|x_1|, \ldots, |x_k|)$.

6 RATIONAL CONSTRAINTS WITH SYNCHRONISATION PARAMETERS

In order to simplify the decision procedure for SL, which we will present in Section 7, we introduce an enriched syntax of rational constraints. We will then extend the AC decision procedure from Section 5 to the new type of constraints such that it can later be used as a subroutine in our decision procedure of SL. Before giving details, we will outline the main idea behind the extension.

The AC decision procedure expects acyclicity, which prohibits formulae that are, e.g., of the form $(\varphi(x) \wedge \varphi'(y)) \wedge \psi(x, y)$. Indeed, after replacing the inner-most conjunction by an equivalent rational constraint, the formula turns into the conjunction $\mathcal{R}_{\varphi \wedge \varphi'}(x, y) \wedge \mathcal{R}_{\psi}(x, y)$, which is a conjunction of the form $\mathcal{R}(x, y) \wedge \mathcal{S}(x, y)$. In general, satisfiability of such conjunctions is not decidable, and they cannot be expressed as a single AFT since synchronisation of ϵ-moves on multiple tracks is not always possible. However, our example conjunction does not compose two arbitrary AFTs. By its construction, $\mathcal{R}_{\varphi \wedge \varphi'}(x, y)$ actually consists of two disjoint AFT parts. Each of the parts constrains symbols read on one of the two tracks only and is completely oblivious of the other part. Due to this, an AFT equivalent to $\mathcal{R}_{\varphi \wedge \varphi'}(x, y) \wedge \mathcal{R}_{\psi}(x, y)$ can be constructed (let us outline, without so far going into details, that the construction would saturate ϵ-moves for each track of $\mathcal{R}_{\varphi \wedge \varphi'}$ separately). Indeed, the original formula can also be rewritten as $\varphi(x) \wedge (\varphi(y) \wedge \psi(x, y))$, which is AC and can be solved by the algorithm of Section 5.

The idea of exploiting the independence of tracks within a transducer can be taken a step further. The two independent parts do not have to be totally oblivious of each other, as in the case of $\mathcal{R}_{\varphi \wedge \varphi'}$ above, but can communicate in a certain limited way. To define the allowed form of communication and to make the independent communicating parts syntactically explicit within string formulae, we will introduce the notion of synchronisation parameters of AFTs. We will then explain how formulae built from constraints with synchronisation parameters can be transformed into a single rational constraint with parameters by a simple adaptation of the AC algorithm, and how the parameters can be subsequently eliminated, leading to a single standard rational constraint.

Definition 6.1 (AFT with synchronisation parameters). An *AFT with parameters* $\bar{s} = s_1, \ldots, s_n$ is defined as a standard AFT $\mathcal{R} = (V, Q, \Delta, I, F)$ with the difference that the initial and the final formula can talk apart from states about so-called *synchronisation parameters* too. That is, $I, F \subseteq \mathbb{F}_{Q \cup \{\bar{s}\}}$ where I is still positive on states and F is still negative on states, but the synchronisation parameters can appear in I and F both positively as well as negatively. The synchronisation parameters put an additional constraint on accepting runs. A run $\rho = \rho_0 \ldots \rho_m$ over a k-tuple of words $\bar{w}$ is accepting

only if there is a truth assignment $\nu : \{\bar{s}\} \to \mathbb{B}$ of parameters such that $\nu \models I$ and $\nu \models F$. We then say that $\bar{w}$ is *accepted with the parameter assignment* ν.

String formulae can be built on top of AFTs with parameters in the same way as before. We write $\varphi[\bar{s}](\bar{x})$ to denote a string formula that uses AFTs with synchronisation parameters from $\bar{s}$ in its rational constraints. Such a formula is interpreted over a union $\iota \cup \nu$ of an assignment $\iota : \mathrm{var}(\varphi) \to \mathcal{P}(V)^*$ from string variables to strings, as usual, and a parameter assignment $\nu : \{\bar{s}\} \to \mathbb{B}$. An atomic constraint $\mathcal{R}[\bar{s}](\bar{x})$ is satisfied by $\iota \cup \nu$, written $\iota \cup \nu \models \mathcal{R}[\bar{s}](\bar{x})$, if $\mathcal{R}$ accepts $(\iota(x_1), \ldots, \iota(x_{|\bar{x}|}))$ with the parameter assignment ν. Atomic string constraints without parameters are satisfied by $\iota \cup \nu$ iff they are satisfied by ι. The satisfaction $\iota \cup \nu \models \varphi$ of a Boolean combination φ of atomic constraints is then defined as usual.

Notice that within a non-trivial string formula, parameters may be shared among AFTs of several rational constraints. They then not only synchronise initial and final configuration of a single transducer run, but provide the aforementioned limited way of communication among AFTs of the rational constraints within the formula.

*Definition 6.2 (*AC *with synchronisation parameters—*ACsp*).* The definition of AC extends quite straightforwardly to rational constraints with parameters. There is no other change in the definition except for allowing rational constraints to use synchronisation parameters as defined above.

Notice that since we do not consider regular constraints with parameters, constraints with parameters in ACsp formulae are never negated.

The synchronisation parameters allow for an easier transformation of string formulae into AC. For instance, consider a formula of the form $\varphi(x, y) \wedge \psi(x, y)$ where one of the conjuncts, say φ, can be rewritten as $\varphi_1[\bar{s}_1](x) \wedge \varphi_2[\bar{s}_2](y)$. The whole formula can be written as $\varphi_1[\bar{s}_1](x) \wedge (\varphi_2[\bar{s}_2](y) \wedge \psi(x, y))$, which falls into ACsp. An example of such a formula $\varphi(x, y)$, commonly found in the benchmarks we experimented with as presented later on, is a formula saying that $x \circ y$ belongs to a regular language, expressed by an AFA $\mathcal{A}$. This can be easily expressed by a conjunction $\mathcal{R}_1[\bar{s}](x) \wedge \mathcal{R}_2[\bar{s}](y)$ of two unary rational constraints with parameters. Intuitively, the AFTs $\mathcal{R}_1$ and $\mathcal{R}_2$ are two copies of $\mathcal{A}$. $\mathcal{R}_1$ nondeterministicaly chooses a configuration where the prefix of a run of $\mathcal{A}$ reading a word x ends, accepts, and remembers the accepting configuration in parameter values (it will have a parameter per state). $\mathcal{R}_2$ then reads the suffix of x, using the information contained in parameter values to start from the configuration where $\mathcal{R}_1$ ended. We explain this construction in detail in Section 7.

An ACsp formula φ with parameters can be translated into a single, parameter-free, rational constraint and then decided by an AFA language emptiness check described in Section 8. The translation is done in two steps:

(1) A **generalised** AC **algorithm** translates $\varphi(\bar{x})$ to $\mathcal{R}_\varphi[\bar{s}](\bar{x})$.
(2) **Parameter elimination** transforms $\mathcal{R}_\varphi[\bar{s}](\bar{x})$ to a normal rational constraint $\mathcal{R}'_\varphi(\bar{x})$.

Generalised AC *algorithm.* To enable eliminations of conjunctions and disjunctions from ACsp formulae, just a small modification of the procedure from Section 5 is enough. The presence of parameters in the initial and final formulae does not require any special treatment, except that, unlike for states (which are implicitly renamed), it is important that sets of synchronisation parameters stay the same even if they intersect, so that the synchronisation is preserved in the resulting AFT. That is, for $\Box \in \{\wedge, \vee\}$, $\mathcal{R}_\varphi[\bar{r}](\bar{x})$, and $\mathcal{R}_\psi[\bar{s}](\bar{y})$, the constraint $\mathcal{R}_{\varphi\Box\psi}[\bar{t}](\bar{z})$ is created in the same way as described in Section 5, the parameters within the initial and the final formulae of the input AFTs are passed to the AFA construction $\Box$ unchanged, and $\{\bar{t}\} = \{\bar{r}\} \cup \{\bar{s}\}$.

Lemma 6.3. *$\mathcal{R}_\varphi[\bar{r}](\bar{x}) \Box \mathcal{R}_\psi[\bar{s}](\bar{y})$ is equivalent to $\mathcal{R}_{\varphi\Box\psi}[\bar{t}](\bar{z})$.*

Elimination of parameters. The previous steps transform the formula into a single rational constraint with synchronisation parameters. Within such a constraint, every parameter communicates one bit of information between the initial and final configuration of a run. The bit can be encoded by an additional automata state passed from a configuration to a configuration via transitions through the entire run, starting from an initial configuration where the parameter value is decided in accordance with the initial formula, to the final configuration where it is checked against the final formula. A technical complication, however, is that automata transitions are monotonic (positive on states). Hence, they cannot prevent arbitrary states from appearing in target configurations even though their presence is not enforced by the source configuration. For instance, starting from a single state q_1 and executing a transition $\Delta(q_1) = q_2$ can yield a configuration $q_2 \wedge q_3$. The assignment of 0 to a parameter cannot therefore be passed through the run in the form of absence of a single designated state as it can be overwritten anywhere during the run.

To circumvent the above, we use a so-called *two rail encoding* of parameter values: every parameter s is encoded using a pair of value indicator states, the positive value indicator s^+ and the negative value indicator s^-. Addition of unnecessary states into target configurations during a run then cannot cause that a parameter silently changes its value. One of the indicators can still get unnecessarily set, but the other indicator will stay in the configuration too (states can be added into the configurations reached, but cannot be removed). The parameter value thus becomes ambiguous—both s^- and s^+ are present. The negative final formula can exclude all runs which arrive with ambiguous parameters by enforcing that at least one of the indicators is false.

Formally, the parameter elimination replaces a constraint $\mathcal{R}(\bar{x})[\bar{s}]$ with $\mathcal{R} = (W\langle|\bar{x}|\rangle, Q, \Delta, I, F)$ and $|\bar{s}| = n$ by a parameter free constraint $\mathcal{R}'(\bar{x})$ with $\mathcal{R}' = (W\langle|\bar{x}|\rangle, Q', \Delta', I', F')$ where

- $Q' = Q \cup \{s_i^+, s_i^- \mid 1 \leq i \leq n\}$ (parameters are added to Q), and
- $\Delta' = \Delta \cup \{s_i^+ \mapsto s_i^+, s_i^- \mapsto s_i^- \mid 1 \leq i \leq n\}$ (once active value indicators stay active).
- $I' = I^+ \wedge \mathit{Choose}$ where I^+ is a positive formula that arises from I by replacing every negative occurrence of a parameter $\neg s$ by a positive occurrence of its negative indicator s^-, and the positive formula $\mathit{Choose} = \bigwedge_{i=1}^{n} s_i^+ \vee s_i^-$ enforces that every parameter has a value.
- $F' = F^- \wedge \mathit{Disambiguate}$ where F^- is a negative formula that arises from F by replacing every positive occurence of a parameter s by a negative occurrence of its negative indicator $\neg s^-$, and the negative formula $\mathit{Disambiguate} = \bigwedge_{i=1}^{n} \neg s_i^+ \vee \neg s_i^-$ enforces that indicators determine parameter values unambiguously, i.e., at most one indicator per parameter is set.

Lemma 6.4. *$\exists \bar{s} : \mathcal{R}(\bar{x})[\bar{s}]$ is equivalent to $\mathcal{R}'(\bar{x})$.*

7 DECIDING STRAIGHT-LINE FORMULAE

Our algorithm solves string formulae using the DPLL(T) framework [Nieuwenhuis et al. 2004][6], where T is a sound and complete solver for AC and SL. Loosely speaking, DPLL(T) can be construed as a collaboration between a DPLL-based SAT-solver and theory solvers, wherein the input formula is viewed as a Boolean formula by the SAT solver, checked for satisfiability by the SAT-solver, and if satisfiable, theory solvers are invoked to check if the Boolean assignment found by the SAT solver can in fact be realised in the involved theories. The details of the DPLL(T) framework are not so important for our purpose. However, the crucial point is that all queries that a DPLL(T) solver asks a T-theory solver are conjunctions from the CNF of the input formula (or their parts), enabling us to concentrate on solving SL conjunctions only.

Our decision procedure for SL conjunctions transforms the input SL conjunction into an equisatisfiable ACsp formula, which is then decided as discussed in Section 6. The rest of the section is thus devoted to a translation of a positive SL conjunction φ to an ACsp formula. The translation

[6] Also see [Kroening and Strichman 2008] for a gentle introduction to DPLL(T).

internally combines rational constraints and equations into a more general kind of constraints in which rational relations are mixed with concatenations and synchronisation parameters.

Example 7.1. As a running example for the section, we use an SL conjunction that captures the essence of the vulnerability pattern from Example 1.1: A sanitizer is applied on an input string to get rid of symbols c, replacing them by d, hoping that this will prevent a dangerous situation which arises when a symbol d apears in a string somewhere behind c. However, the dangerous situation will not be completely avoided since it is forgotten that the sanitized string will be concatenated with another string that can still contain c.[7]

To formalize the example, assume a bit-vector encoding of an alphabet Σ which contains the symbols c and d. Assume that each $a \in \Sigma$ denotes the conjunction of (negated) bit variables encoding it. As our running example, we will then consider the formula $\varphi : y = \mathcal{R}(x) \wedge z = x \circ y \wedge \mathcal{A}(z)$. The AFT $\mathcal{R} = (W\langle 2\rangle, Q = \{q\}, \Delta = \{q \mapsto q \wedge \neg \mathsf{d}^1 \wedge (\mathsf{c}^1 \rightarrow \mathsf{d}^2) \wedge \bigwedge_{a \in \Sigma \setminus \{\mathsf{c}\}} (a^1 \leftrightarrow a^2))\}, I = q, F = \mathtt{true})$ is a sanitizer that produces y by replacing all occurrences of c in its input string x by d, and it also makes sure that x does not include d. The AFA $\mathcal{A} = (V, Q' = \{r_0, r_1, r_2\}, \Delta', I' = r_0, F' = \neg r_0 \wedge \neg r_1)$ where $\Delta'(r_0) = (r_0 \wedge \neg \mathsf{c}) \vee (r_1 \wedge \mathsf{c})$, $\Delta'(r_1) = (r_1 \wedge \neg \mathsf{d}) \vee (r_2 \wedge \mathsf{d})$, and $\Delta'(r_2) = \mathtt{true}$ is the specification. It checks whether the opening symbol c can be later followed by the closing symbol d in the string z. The formula is satisfiable. □

Definition 7.2 (Mixed constraints). A *mixed constraint* is of the form $x = \mathcal{R}[\bar{s}](y_1 \circ \cdots \circ y_n)$ where $\mathcal{R}$ is a binary AFT, with a concatenation of variables as the right-hand side argument, and $\bar{s}$ is a vector of synchronisation parameters. Such constraint has the expected meaning: it is satisfied by the union $\nu \cup \iota$ of an assignment ι to string variables and an assignment ν to parameters iff $(\iota(x), \iota(y_1) \circ \cdots \circ \iota(y_n))$ is accepted by $\mathcal{R}[\bar{s}]$ with the parameter assignment ν.

All steps of our translation of the input SL formula φ to an ACsp formula preserve the SL fragment, naturally generalised to mixed constraints as follows.

Definition 7.3 (Generalised straight-line conjunction). A conjunction of string constraints is defined to be generalised *straight-line* if it can be written as $\psi \wedge \bigwedge_{i=1}^{m} x_i = F_i$ where ψ is a conjunction over regular and negated regular constraints and each F_i is either of the form $y_1 \circ \cdots \circ y_n$ or $\mathcal{R}[\bar{s}](y_1 \circ \cdots \circ y_n)$ such that it does not contain variables $x_i, \ldots, x_m$.

For simplicity, we assume that φ has gone through two preprocessing steps. First, all negations were eliminated by complementing regular constraints, resulting in a purely positive conjunction. Second, all the—now only positive—regular constraints were replaced by equivalent rational constraints. Particularly, a regular constraint $\mathcal{A}(x)$ is replaced by a rational constraint $x' = \mathcal{R}'(x)$ where x' is a fresh variable and $\mathcal{R}'$ is an AFT with $Rel(\mathcal{R}') = \mathcal{P}(V)^* \times L(\mathcal{A})$. The AFT $\mathcal{R}'$ is created from $\mathcal{A}$ by indexing all propositions in the transition relation by the index 2 of the second track. It is not difficult to see that since x' is fresh, the replacement preserves SL, and also satisfiability, since $P(x) \wedge \psi$ is equivalent to $\exists x' : x' = \mathcal{R}(x) \wedge \psi$ for every ψ.

Example 7.4. In Example 7.1, the preprocessing replaces the conjunct $\mathcal{A}(z)$ by $z' = \mathcal{S}(z)$ where $\mathcal{S}$ is the same as $\mathcal{A}$, except occurrences of bit-vector variables in Δ' are indexed by 2 since z will be read on its second track. We obtain $\varphi_0' : y = \mathcal{R}(x) \wedge z = x \circ y \wedge z' = \mathcal{S}(z)$ where z' is free. □

Due to the preprocessing, we are starting with a formula φ_0' in the form of an SL conjunction of rational constraints and equations. The translation to ACsp will be carried out in the following three steps, which will be detailed in the rest of the section:

[7] In reality, where one undesirably concatenates a string `command('...` with some string `...');attack();` the situation is, of course, more complex and sanitization is more sophisticated. However, having a real-life example, such as those used in our experiments, as a running example would be too complex to understand.

(1) **Substitution** transforms φ_0' to a conjunction φ_1 of mixed constraints.
(2) **Splitting** transforms φ_1 to a conjunction φ_2 of rational constraints with parameters.
(3) **Ordering** transforms φ_2 to an AC conjunction φ_3 with parameters.

Substitution. Equations in φ_0' are combined with rational constraints into mixed constraints by a straightforward substitution. In one substitution step, a conjunction $x = y_1 \circ \cdots \circ y_n \wedge \psi$ is replaced by $\psi[y_1 \circ \cdots \circ y_n/x]$ where all occurrences of x are replaced by $y_1 \circ \cdots \circ y_n$. The substitution preserves the generalised straight-line fragment.

Lemma 7.5. *If $x = y_1 \circ \cdots \circ y_n \wedge \psi$ is* SL*, then $\psi[y_1 \circ \cdots \circ y_n/x]$ is equisatisfiable and* SL.

The substitution steps are iterated eagerly in an arbitrary order until there are no equations. Every substitution step obviously decreases the number of equations, so the iterative process terminates after a finitely many steps with an equation-free SL conjunction of mixed constraints φ_1.

Example 7.6. The substitution eliminates the equation $z = x \circ y$ in φ_0' from Example 7.4, transforming it to $\varphi_1 : y = \mathcal{R}(x) \wedge u = \mathcal{S}(x \circ y)$. □

Splitting. We will now explain how synchronisation parameters are used to eliminate concatenation within mixed constraints. The operation of *binary splitting* applied to an SL conjunction of mixed constraints, $\varphi : x = \mathcal{R}(y_1 \circ \cdots \circ y_m \ \circ \ z_1 \circ \cdots \circ z_n)[\bar{s}] \wedge \psi$, where $\mathcal{R} = (W\langle 2\rangle, Q, \Delta, I, F)$ and $Q = \{q_1, \ldots, q_l\}$ splits the mixed constraint and substitutes x by a concatenation of fresh variables $x_1 \circ x_2$ in ψ. That is, it outputs the conjunction $\varphi' : \zeta \wedge \psi[x_1 \circ x_2/x]$ of mixed constraints, where the rational constraint was split into the following conjunction ζ of two constraints:

$$\zeta : x_1 = \mathcal{R}_1(y_1 \circ \cdots \circ y_m)[\bar{s}, \bar{t}] \wedge x_2 = \mathcal{R}_2(z_1 \circ \cdots \circ z_n)[\bar{s}, \bar{t}]$$

The vector $\bar{t}$ consists of l fresh parameters, x_1 and x_2 are fresh string variables, and each AFT with parameters $\mathcal{R}_i = (W\langle 2\rangle, Q, \Delta, I_i, F_i), i \in \{1, 2\}$, is derived from $\mathcal{R}$ by choosing initial/final formulae:

$$I_1 = I, \quad F_1 = \bigwedge_{i=1}^{l} q_i \to t_i, \qquad I_2 = \bigwedge_{i=1}^{l} t_i \to q_i, \quad F_2 = F\ .$$

Intuitively, each run ρ of $\mathcal{R}$ is split into a run ρ_1 of $\mathcal{R}_1$, which corresponds to the first part of ρ in which $y_1 \circ \cdots \circ y_m$ is read along with a prefix x_1 of x, and a run ρ_2 of $\mathcal{R}_2$, which corresponds to the part of ρ in which $z_1 \circ \cdots \circ z_n$ is read along with the suffix x_2 of x. Using the new synchronisation parameters $\bar{t}$, the formulae F_1 and I_2 ensure that the run ρ_1 of $\mathcal{R}_1$ must indeed start in the states in which the run ρ_2 of $\mathcal{R}_2$ ended, that is, the original run ρ of $\mathcal{R}$ can be reconstructed by connecting ρ_1 and ρ_2. Every occurrence of x in ψ is replaced by the concatenation $x_1 \circ x_2$.

Lemma 7.7. *In the above, φ is equivalent to $\exists x_1 x_2 \bar{t} : \varphi'$.*

The resulting formula φ' is hence equisatisfiable to the original φ. Moreover, φ' is still generalised SL—the two new constraints defining x_1 and x_2 can be placed at the position of the original constraint defining x that was split, and the substitution $[x_1 \circ x_2/x]$ in the rest of the formula only applies to the right-hand sides of constraints (since x can be defined only once).

Lemma 7.8. *If φ is an* SL *conjunction of mixed constraints, then so is φ'.*

Moreover, by applying binary splitting steps eagerly in an arbitrary order on φ_1, we are guaranteed that all concatenations will be eliminated after a finite number of steps, thus arriving at the SL conjunction of rational constraints with parameters φ_2. The termination argument relies on the straight-line restriction. Although it cannot be simply said that every step reduces the number of concatenations because the substitution $x_1 \circ x_2$ introduces new ones, the new concatenations $x_1 \circ x_2$ are introduced only into constraints defining variables that are higher in the straight-line

ordering than x. It is therefore possible to define a well-founded (integer) measure on the formulae that decreases with every application of the binary splitting steps.

Lemma 7.9. *All concatenations in the* SL *conjunction of mixed constraints φ_1 will be eliminated after a finite number of binary splitting steps.*

We note that our implementation actually uses a slightly more efficient n-ary splitting instead of the described binary. It splits a mixed constraint in one step into the number of conjuncts equal to the length of the concatenation in its right-hand side. We present the simpler binary variant, which eventually achieves the same effect.

Example 7.10. The formula from Example 7.6 would be transformed into $\varphi_2 : y = \mathcal{R}(x) \wedge u_1 = \mathcal{S}_1[\bar{s}](x) \wedge \mathcal{S}_2[\bar{s}](y)$ where $\mathcal{S}_1, \mathcal{S}_2$ are as $\mathcal{S}$ up to that $\mathcal{S}_1$ has the final formula $I' \wedge \bigwedge_{i=0}^{2}(r_i \to s_0)$ and $\mathcal{S}_2$ has the final formula $F' \wedge \bigwedge_{i=0}^{2}(s_i \to r_i)$. Notice that $u_1 = \mathcal{S}_1[\bar{s}](x) \wedge u_2 = \mathcal{S}_2[\bar{s}](y)$ still enforce that $x \circ y$ has c eventually followed by d. The parameters remember where $\mathcal{S}_1$ ended its run and force $\mathcal{R}_2$ to continue from the same state. □

Reordering modulo associativity. Substitution and splitting transform φ_0 to a straight-line conjunction φ_2 of rational constraints with parameters. Before delegating it to the ACsp formulae solver, it must be reorganized modulo associativity to achieve a structure satisfying the definition of AC. One way of achieving this is to order the formula into a conjunction $\bigwedge_{i=1}^{m} x_i = \mathcal{R}[\bar{s}^i](y_i)$ satisfying the condition in the definition of SL (the definition of SL only requires that the formula *can* be assumed). An simple way is discussed in [Lin and Barceló 2016]. It consists of drawing the dependency graph of φ, a directed graph with the variables $\mathrm{var}(\varphi)$ as vertices which has an edge $x \to y$ if and only if φ contains a conjunct $x = \mathcal{R}(y)$. Due to the straight-line restriction, the graph must be acyclic. The ordering of variables can be then obtained as a topological sort of the graphs vertices, which is computable in linear time (e.g. [Cormen et al. 2009], for instance by a depth-first traversal). The final acyclic formula φ_3 then arises when letting $\bigwedge_{i=1}^{m}$ associate from the right:

$$\varphi_3 : (x_1 = \mathcal{R}_1(y_1) \wedge (x_2 = \mathcal{R}_2(y_2) \wedge (\ldots \wedge (x_{m-1} = \mathcal{R}_{m-1}(y_{m-1}) \wedge x_m = \mathcal{R}_m(y_m)) \ldots))).$$

To see that φ_3 is indeed ACsp, observe that every conjunctive sub-formula is of the form $(\bigwedge_{i<k} x_i = \mathcal{R}_i(y_i)) \wedge x_k = \mathcal{R}_k(y_k)$ where x_k is by the definition of SL not present in the left conjunct. The left and right conjuncts can therefore share at most one variable, y_k.

Theorem 7.11. *The formula φ_3 obtained by substitution, splitting, and reordering from φ_0 is equisatisfiable and acyclic.*

Example 7.12. The ACsp formula $\varphi_3 : y = \mathcal{R}(x) \wedge u_1 = \mathcal{S}_1[\bar{s}](x)) \wedge \mathcal{S}_2[\bar{s}](y)$ would be the final result of the SL to ACsp translation. Let us use φ_3 to also briefly illustrate the decision procedure for ACsp of Section 6. The first step is the transformation to a single rational constraint with parameters by induction over formula structure. This will produce $\mathcal{R}'[\bar{s}](x, y, z)$ with states and transitions consisting of those in $\mathcal{R}$, $\mathcal{S}_1$ with indexes of alphabet bits incremented by one (y, and z are now not the first and the second, but the second and the third track), and a copy $\mathcal{S}_2'$ of $\mathcal{S}_2$ with states replaced by their primed variant (so that they are disjoint from that of $\mathcal{S}_1$) and also incremented indexes of alphabet bits. The initial and final configuration will be the conjunctions of those of $\mathcal{R}, \mathcal{S}_1$ and $\mathcal{S}_2'$. The last step, eliminating of parameters, will lead to the addition of positive and negative indicator states for parameters $\bar{s} = s_1, s_2, s_3$ with the universal self-loops and the update of the initial and final formula as in Section 6. The rest is solved by the emptiness check discussed in Section 8. Notice the small size of the resulting AFT. Compared to the original formula from Example 7.1, it contains only one additional copy of $\mathcal{A}$ (the $\mathcal{S}_2'$), the six additional parameter indicator states with self-loops and the initial and final condition on the parameter indicators. □

A note on the algorithm of [Lin and Barceló 2016]. We will now comment on the differences of our algorithm for deciding SL from the earlier algorithm of [Lin and Barceló 2016]. It combines reasoning on the level NFAs and nondeterministic transducers, utilising classical automata theoretic techniques, with a technique for eliminating concatenation by enumerative automata splitting. It first turns and SL formula into a pure AC formula and then uses the AC decision procedure.

An obvious advantage of our decision procedure described in Section 5 is the use of succinct AFA. As opposed to the worst case exponentially larger NFA, it produces an AFA of a linear size (unless the original formula contains negated regular constraints represented as general AFA. See Section 5 for a detailed discussion). Let us also emphasize the advantages of our algorithm in the first phase, translation of SL to ACsp. Similarly as in the case of deciding AC, the main advantage of our algorithm is that, while [Lin and Barceló 2016] only works with NFTs, we propose ways of utilising the power of alternation and succinct transition encoding.

We will illustrate the difference on an example. The concatenation in the conjunction $x = y \circ z \wedge w = \mathcal{R}(x)$ would in [Lin and Barceló 2016] be done by enumerative splitting. It replaces the conjunction by the disjunction $\bigvee_{q \in Q} w_1 = \mathcal{R}_q(y) \wedge w_2 = {}_q\mathcal{R}(z)$. The Q in the disjunction is the set of states of the (nondeterministic) transducer $\mathcal{R}$, $\mathcal{R}_q$ is the same as the NFT $\mathcal{R}$ up to that the final state is q, and ${}_q\mathcal{R}$ the same as $\mathcal{R}$ up to that the initial state is q. Intuitively, the run of $\mathcal{R}$ is explicitly separated into the part in which y is read along the prefix w_1 of w, and the suffix in which z is red along the suffix w_2 of w. The variable w would be replaced by $w_1 \circ w_2$ in the rest of the formula. The disjunction enumerates all admissible intermediate states $q \in Q$ a run of $\mathcal{R}$ can cross, and for each of them, it constructs two copies of $\mathcal{R}$. This makes the cost of the transformation quadratic in the number of states of the NFT $\mathcal{R}$. A straightforward generalisation to our setting in which $\mathcal{R}$ is an AFT is possible: The disjunction would have to list, instead of possible intermediate states $q \in Q$, all possible intermediate configurations $C \subseteq Q$ a run of the AFA $\mathcal{R}$ can cross, thus increasing the quadratic blow-up of the nondeterministic case to an exponential (due to the enumerative nature of splitting, the size is without any optimisation bounded by an exponential even from below).

Our splitting algorithm utilises succinctness of alternation to reduce the cost of enumerative AFA splitting from exponential space (or quadratic in the case of NFAs) to linear. The smaller size of the resulting representation is payed for by a more complex alternating structure of the resulting rational constraints. The worst case complexity of the satisfiability procedure thus remains essentially the same. However, deferring most of the complexity to the last phase of the decision procedure, AFA emptiness checking, allows to circumvent the potential blow-up by means of modern model checking algorithms and heuristics and achieve much better scalability in practice.

8 MODEL CHECKING FOR AFA LANGUAGE EMPTINESS

In order to check unsatisfiability of a string formula using our translation to AFTs, it is necessary to show that the resulting AFT does not accept any word, i.e., that the recognised language is empty. The constructed AFTs are succinct, but tend to be quite complex: a naïve algorithm that would translate AFTs to NFAs using an explicit subset construction, followed by systematic state-space exploration, is therefore unlikely to scale to realistic string problems. We discuss how the problem of AFT emptiness can instead be reduced (in linear time and space) to reachability in a Boolean transition system, in a way similar to [Cox and Leasure 2017; Gange et al. 2013; Wang et al. 2016]. Our translation is also inspired by the use of model checkers to determinise NFAs in [Tabakov and Vardi 2005], by a translation to sequential circuits that corresponds to symbolic subset construction. We use a similar implicit construction to map AFAs and AFTs to NFAs.

As an efficiency aspect of the construction for AFAs, we observe that it is enough to work with *minimal* sets of states, thanks to the monotonicity properties of AFAs (the fact that initial formulae and transition formulae are positive in the state variables, and final formulae are negative). This

gives rise to three different versions: a direct translation that does not enforce minimality at all; an *intensionally-minimal* translation that only considers minimal sets by virtue of additional Boolean constraints; and a *deterministic* translation that resolves nondeterminism through additional system inputs, but does not ensure fully-minimal state sets.

8.1 Direct Translation to Transition Systems

To simplify the presentation of our translation to a Boolean transition system, we focus on the case of AFAs $\mathcal{A} = (V_n, Q, \Delta, I, F)$ over a single track of bit-vectors of length $n+1$. The translation directly generalises to k-track AFTs, and to AFTs with epsilon characters, by simply choosing n sufficiently large to cover the bits of all tracks.

We adopt a standard Boolean transition system view on the execution of the AFA $\mathcal{A}$ (e.g., [Clarke et al. 1999]). If $\mathcal{A}$ has $m = |Q|$ automaton states, then $\mathcal{A}$ can be interpreted as a (symbolically described) transition system $T^{di}_{\mathcal{A}} = (\mathbb{B}^m, \mathit{Init}^{di}, \mathit{Trans}^{di})$. The transition system has state space $\mathbb{B}^m$, i.e., a system state is a bit-vector $\bar{q} = \langle q_0, \ldots, q_{m-1}\rangle$ of length m identifying the active states in Q. The initial states of the system are defined by $\mathit{Init}^{di}[\bar{q}] = I$, the same positive Boolean formula as in $\mathcal{A}$. The transition relation Trans^{di} of the system is a Boolean formula over two copies $\bar{q}, \bar{q}'$ of the state variables, encoding that for each active pre-state q_i in $\bar{q}$ the formula $\Delta(q_i)$ has to be satisfied by the post-state $\bar{q}'$. Input variables $V_n = \{x_0, ..., x_n\}$ are existentially quantified in the transition formula, expressing that all AFA transitions have to agree on the letter to be read:

$$\mathit{Trans}^{di}[\bar{q}, \bar{q}'] \;=\; \exists v_0, \ldots, v_n : \bigwedge_{i=0}^{m-1} q_i \rightarrow \Delta(q_i)[\bar{q}/\bar{q}'] \tag{1}$$

To examine emptiness of $\mathcal{A}$, it has to be checked whether $T^{di}_{\mathcal{A}}$ can reach any state in the target set $\mathit{Final}^{di}[\bar{q}] = F$, i.e., in the set described by the negative final formula F of $\mathcal{A}$. Since is well-known that reachability in transition systems is a PSPACE-complete problem [Clarke et al. 1999], this directly establishes that fragment AC is in PSPACE (Corollary 5.4).

Lemma 8.1. *The language $L(\mathcal{A})$ recognised by the AFA $\mathcal{A}$ is empty if and only if $T^{di}_{\mathcal{A}}$ cannot reach a configuration in $\mathit{Final}^{di}[\bar{q}]$.*

In practice, this means that emptiness of $L(\mathcal{A})$ can be decided using a wide range of readily available, highly optimised model checkers from the hardware verification field, utilising methods such as k-induction [Sheeran et al. 2000], Craig interpolation [McMillan 2003], or IC3/PDR [Bradley 2012]. In our implementation, we represent $T^{di}_{\mathcal{A}}$ in the AIGER format [Biere et al. 2017], and then apply nuXmv [Cavada et al. 2014] and ABC [Brayton and Mishchenko 2010].

The encoding $T^{di}_{\mathcal{A}}$ leaves room for optimisation, however, as it does not fully exploit the structure of AFAs and introduces more transitions than strictly necessary. In (1), we can observe that if $\mathit{Trans}^{di}[\bar{q}, \bar{q}']$ is satisfied for some $\bar{q}, \bar{q}'$, then it will also be satisfied for every post-state $\bar{q}'' \succeq \bar{q}'$, writing $\bar{p} \preceq \bar{q}$ for the point-wise order on bit-vectors $\bar{p}, \bar{q} \in \mathbb{B}^m$ (i.e., $\bar{p} \preceq \bar{q}$ if p_i implies q_i for every $i \in \{0, \ldots, m-1\}$). This is due to the positiveness (or *monotonicity*) of the transition formulae $\Delta(q_i)$. Similarly, since the initial formula I of an AFA is positive, initially more states than necessary might be activated. Because the final formula F is negative, and since redundant active states can only impose additional restrictions on the possible runs of an AFA, such redundant states can never lead to more words being accepted.

More formally, we can observe that the transition system $T^{di}_{\mathcal{A}}$ is *well-structured* [Finkel 1987], which means that the state space $\mathbb{B}^m$ can be equipped with a well-quasi-order $\leq$ such that whenever $\mathit{Trans}^{di}[\bar{q}, \bar{q}']$ and $\bar{q} \leq \bar{p}$, then there is some state $\bar{p}'$ with $\bar{q}' \leq \bar{p}'$ and $\mathit{Trans}^{di}[\bar{p}, \bar{p}']$. In our case, $\leq$ is

the inverse point-wise order $\geq$ on bit-vectors;[8] intuitively, deactivating AFA states can only enable more transitions. Since the set $Final^{di}[\bar{q}]$ is upward-closed with respect to $\leq$ (downward-closed with respect to $\preceq$), the theory on well-structured transition systems tells us that it is enough to consider transitions to $\leq$-maximal states (or $\preceq$-minimal states) of the transition system when checking reachability of $Final^{di}[\bar{q}]$. In forward-exploration of the reachable states of $T^{di}_{\mathcal{A}}$, the non-redundant states to be considered form an anti-chain. This can be exploited by defining tailor-made exploration algorithms [Doyen and Raskin 2010; Kloos et al. 2013], or, as done in the next sections, by modifying the transition system to only include non-redundant transitions.

8.2 Intensionally-Minimal Translation

We introduce several restricted versions of the transition system $T^{di}_{\mathcal{A}}$, by removing transitions to non-minimal states. The strongest transition system $T^{\min}_{\mathcal{A}} = (\mathbb{B}^m, Init^{\min}, Trans^{\min})$ obtained in this way can abstractly be defined as:

$$Init^{\min}[\bar{q}] \;=\; Init^{di}[\bar{q}] \wedge \forall \bar{p} \prec \bar{q}.\ \neg Init^{di}[\bar{p}] \tag{2}$$

$$Trans^{\min}[\bar{q}, \bar{q}'] \;=\; Trans^{di}[\bar{q}, \bar{q}'] \wedge \forall \bar{p} \prec \bar{q}'.\ \neg Trans^{di}[\bar{q}, \bar{p}] \tag{3}$$

That means, $Init^{\min}$ and $Trans^{\min}$ are defined to only retain the $\preceq$-minimal states. Computing $Init^{\min}$ and $Trans^{\min}$ corresponds to the logical problem of *circumscription* [McCarthy 1980], i.e., the computation of the set of minimal models of a formula. Circumscription is in general computationally hard, and its precise complexity still open in many cases; in (2) and (3), note that eliminating the universal quantifiers (as well as the universal quantifiers introduced by negation of $Trans^{di}$) might lead to an exponential increase in formula size, so that $T^{\min}_{\mathcal{A}}$ does not directly appear useful as input to a model checker.

We can derive a more practical, but weaker system $T^{\mathrm{im}}_{\mathcal{A}} = (\mathbb{B}^m, Init^{\mathrm{im}}, Trans^{\mathrm{im}})$ by only minimising post-states in $Trans^{\mathrm{im}}$ with respect to the same input letter V_n:

$$\begin{aligned} Init^{\mathrm{im}}[\bar{q}] \;&=\; Init^{\min}[\bar{q}] \\ Trans^{\mathrm{im}}[\bar{q}, \bar{q}'] \;&=\; \exists V_n.\ \Big(Trans[\bar{q}, \bar{q}', V_n] \wedge \forall \bar{p} \prec \bar{q}'.\ \neg Trans[\bar{q}, \bar{p}, V_n] \Big) \\ \text{with} \quad Trans[\bar{q}, \bar{q}', V_n] \;&=\; \bigwedge_{i=0}^{m-1} q_i \rightarrow \Delta(q_i)[\bar{q}/\bar{q}'] \end{aligned}$$

The formulae still contain universal quantifiers $\forall \bar{p}$, but it turns out that the quantifiers can now be eliminated with only polynomial effort, due to the fact that $\bar{p}$ only occurs negatively in the scope of the quantifier. Indeed, whenever $\varphi[\bar{q}]$ is a formula that is positive in $\bar{q}$, and $\varphi[\bar{q}]$ holds for assignments $\bar{q}_1, \bar{q}_3 \in \mathbb{B}^m$ with $\bar{q}_1 \preceq \bar{q}_3$, then $\varphi[\bar{q}]$ will also hold for any assignment $\bar{q}_2 \in \mathbb{B}^m$ with $\bar{q}_1 \preceq \bar{q}_2 \preceq \bar{q}_3$ due to monotonicity. This implies that a satisfying assignment $\bar{q}_1 \in \mathbb{B}^m$ is $\preceq$-minimal if no single bit in $\bar{q}_1$ can be switched from 1 to 0 without violating $\varphi[\bar{q}]$. More formally, $\varphi[\bar{q}] \wedge \neg\exists \bar{p} \prec \bar{q}.\ \varphi[\bar{p}]$ is equivalent to $\varphi[\bar{q}] \wedge \bigwedge_{i=0}^{m-1} q_i \rightarrow \neg\varphi[\bar{q}][q_i/\texttt{false}]$, where we write $\varphi[q_i/\texttt{false}]$ for the result of substituting q_i with `false` in φ.

[8] Since the state space $\mathbb{B}^m$ of $T^{di}_{\mathcal{A}}$ is finite, the "well-" part is trivial.

The corresponding, purely existential representation of $Init^{\text{im}}$ and $Trans^{\text{im}}$ is:

$$Init^{\text{im}}[\bar{q}] \;\equiv\; Init^{di}[\bar{q}] \wedge \bigwedge_{i=0}^{m-1} q_i \rightarrow \neg Init^{di}[\bar{q}][q_i/\texttt{false}] \tag{4}$$

$$Trans^{\text{im}}[\bar{q},\bar{q}'] \;\equiv\; \exists V_n.\, \left(Trans[\bar{q},\bar{q}',V_n] \wedge \bigwedge_{i=0}^{m-1} q_i' \rightarrow \neg Trans[\bar{q},\bar{q}',V_n][q_i'/\texttt{false}]\right) \tag{5}$$

The representation is quadratic in size of the original formulae $Init^{di}$, $Trans^{di}$, but the formulae can in practice be reduced drastically by sharing of common sub-formulae, since the m copies of $Init^{di}[\bar{q}][q_i/\texttt{false}]$ and $Trans[\bar{q},\bar{q}',V_n][q_i'/\texttt{false}]$ tend to be almost identical.

Lemma 8.2. *The following statements are equivalent:*

(1) $T^{di}_{\mathcal{A}}$ can reach a configuration in $Final^{di}[\bar{q}]$;
(2) $T^{min}_{\mathcal{A}}$ can reach a configuration in $Final^{di}[\bar{q}]$;
(3) $T^{im}_{\mathcal{A}}$ can reach a configuration in $Final^{di}[\bar{q}]$.

Example 8.3. To illustrate the $T^{im}_{\mathcal{A}}$ encoding, we consider an AFA $\mathcal{A}$ that accepts the language $\{xwy \mid |xwy| = 2k, k \geq 1, x \in \{a,b\}, y \in \{c,d\}\}$ using the encoding of the alphabet $\Sigma = \{a,b,c,d\}$ from Example 4.1. We let $\mathcal{A} = (\{v_0,v_1\},\{q_0,q_1,q_2,q_3,q_4\},\Delta,I,F)$ where $I = q_0$, $F = \neg q_0 \wedge \neg q_1 \wedge \neg q_3$ (i.e., the accepting states are q_2 and q_4), and Δ is defined as $\Delta(q_0) = \neg v_1 \wedge q_1 \wedge q_3$, $\Delta(q_1) = q_2$, $\Delta(q_2) = q_1$, $\Delta(q_3) = q_3 \;\vee\; (v_1 \wedge q_4)$, and $\Delta(q_4) = \texttt{false}$.

The direct transition system representation is $T^{di}_{\mathcal{A}} = (\mathbb{B}^5, Init^{di}, Trans^{di})$, defined by:

$$Init^{di}[\bar{q}] \;=\; q_0, \qquad Trans^{di}[\bar{q},\bar{q}'] \;=\; \exists v_0,v_1.\, \underbrace{\begin{pmatrix}(q_0 \rightarrow \neg v_1 \wedge q_1' \wedge q_3') \wedge \\ (q_1 \rightarrow q_2') \wedge \\ (q_2 \rightarrow q_1') \wedge \\ (q_3 \rightarrow q_3' \;\vee\; (v_1 \wedge q_4')) \wedge \\ (q_4 \rightarrow \texttt{false})\end{pmatrix}}_{Trans[\bar{q},\bar{q}',V_n]}$$

The intensionally-minimal translation $T^{im}_{\mathcal{A}}$ can be derived from $T^{di}_{\mathcal{A}}$ by conjoining the restrictions in (4) and (5) ($Trans^{im}[\bar{q},\bar{q}']$ is shown in simplified form for sake of presentation):

$$Init^{im}[\bar{q}] \;=\; q_0 \wedge (q_0 \rightarrow \neg\texttt{false}) \wedge \bigwedge_{i=1}^{4} (q_i \rightarrow \neg q_0) \;\equiv\; q_0 \wedge \neg q_1 \wedge \neg q_2 \wedge \neg q_3 \wedge \neg q_4$$

$$Trans^{im}[\bar{q},\bar{q}'] \;\equiv\; \exists v_0,v_1.\, \begin{pmatrix}Trans[\bar{q},\bar{q}',V_n] \wedge \neg q_0' \wedge (q_1' \rightarrow q_0 \vee q_2) \wedge (q_2' \rightarrow q_1) \wedge \\ (q_3' \rightarrow q_0 \vee (q_3 \wedge \neg(v_1 \wedge q_4'))) \wedge (q_4' \rightarrow q_3 \wedge \neg q_3')\end{pmatrix}$$

□

8.3 Deterministic Translation

We introduce a further encoding of $\mathcal{A}$ as a transition system that is more compact than (4), (5), but does not always ensure fully-minimal state sets. The main idea of the encoding is that a conjunctive transition formula $\Delta(q_1) = q_2 \wedge q_3$, assuming that q_2, q_3 do not occur in any other transition formula $\Delta(q_i)$, can be interpreted as a set of deterministic updates $q_2' = q_1; q_3' = q_1$. For state variables that occur in multiple transition formulae, the right-hand side of the update turns into a disjunction. Disjunctions in transition formulae represent nondeterministic updates that can be resolved using additional Boolean flags. The resulting transition system is deterministic, as transitions are uniquely determined by the pre-state and variables representing system inputs.

Example 8.4. We illustrate the encoding $T^{det}_{\mathcal{A}} = (\mathbb{B}^m, Init^{det}, Trans^{det})$ using the AFA from Example 8.3. While the initial states $Init^{det}[\bar{q}]$ coincide with $Init^{im}[\bar{q}]$ in Example 8.3, the transition relation $Trans^{det}[\bar{q}, \bar{q}']$ now consists of two parts: a deterministic assignment of the post-state $\bar{q}'$ in terms of the pre-state $\bar{q}$, together with an auxiliary variable h_3 that determines which branch of $\Delta(q_3)$ is taken; and a conjunct that ensures that value of h_3 is consistent with the inputs V_n. The resulting $Trans^{det}[\bar{q}, \bar{q}']$ is (in this example) equivalent to $Trans^{im}[\bar{q}, \bar{q}']$:

$$
\begin{aligned}
Init^{det}[\bar{q}] \;&=\; q_0 \wedge \neg q_1 \wedge \neg q_2 \wedge \neg q_3 \wedge \neg q_4 \\
Trans^{det}[\bar{q}, \bar{q}'] \;&\equiv\; \exists h_3.\, \begin{pmatrix} (q_0' \leftrightarrow \texttt{false}) \wedge \\ (q_1' \leftrightarrow q_0 \vee q_2) \wedge \\ (q_2' \leftrightarrow q_1) \wedge \\ (q_3' \leftrightarrow q_0 \vee q_3 \wedge h_3) \wedge \\ (q_4' \leftrightarrow q_3 \wedge \neg h_3) \end{pmatrix} \wedge \exists v_0, v_1.\, \begin{pmatrix} (q_0 \rightarrow \neg v_1) \wedge \\ (q_3 \wedge \neg h_3 \rightarrow v_1) \wedge \\ (q_4 \rightarrow \texttt{false}) \end{pmatrix}
\end{aligned}
$$

□

To define the encoding formally, we make the simplifying assumption that there is a unique initial state q_0, i.e., $I = q_0$, and that all transition formulae $\Delta(q_i)$ are in negation normal form (i.e., in particular state variables in $\Delta(q_i)$ do not occur underneath negation). Both assumption can be established by simple transformation of $\mathcal{A}$. The transition system $T^{det}_{\mathcal{A}} = (\mathbb{B}^m, Init^{det}, Trans^{det})$ is:

$$
\begin{aligned}
Init^{\mathrm{det}}[\bar{q}] \;&=\; q_0 \wedge \bigwedge_{i=1}^{m-1} \neg q_i \\
Trans^{\mathrm{det}}[\bar{q}, \bar{q}'] \;&=\; \exists H.\, \left(\left(\bigwedge_{i=0}^{m-1} q_i' \leftrightarrow NewState(q_i) \right) \wedge \exists V_n.\, \left(\bigwedge_{i=0}^{m-1} q_i \rightarrow InputInv(\Delta(q_i), i) \right) \right)
\end{aligned}
$$

The transition relation $Trans^{\mathrm{det}}$ consists of two main parts: the state updates, which assert that every post-state variable q_i' is set to an update formula $NewState(q_i)$; and an input invariant asserting that the letters that are read are consistent with the transition taken. To determinise disjunctions in transition formulae $\Delta(q_i)$, a set H of additional Boolean variables h_l (uniquely indexed by a position sequence $l \in \mathbb{Z}^*$) is introduced, and existentially quantified in $Trans^{\mathrm{det}}$.

The update formulae $NewState(q_i)$ are defined as a disjunction of assignments extracted from the transition formulae $\Delta(q_j)$,

$$
NewState(q_i) \;=\; \bigvee \{ \varphi \mid \text{there is } j \in \{0, \ldots, m-1\} \text{ such that } \langle q_i, \varphi \rangle \in StateAsgn(\Delta(q_j), j, q_j) \}
$$

where each $StateAsgn(\Delta(q_j), j, q_j)$ represents the set of asserted state variables q_i in $\Delta(q_j)$, together with guards φ for the case that q_i occurs underneath disjunctions. The set is recursively defined (on formulae in NNF) as follows:

$$
\begin{aligned}
StateAsgn(\varphi_1 \wedge \varphi_2, l, g) \;&=\; StateAsgn(\varphi_1, l, g) \cup StateAsgn(\varphi_2, l, g) \\
StateAsgn(\varphi_1 \vee \varphi_2, l, g) \;&=\; StateAsgn(\varphi_1, l.1, g \wedge h_l) \cup StateAsgn(\varphi_2, l.2, g \wedge \neg h_l) \\
StateAsgn(q_i, l, g) \;&=\; \{\langle q_i, g \rangle\} \\
StateAsgn(\phi, l, g) \;&=\; \emptyset \qquad \text{(for any other } \phi\text{)} .
\end{aligned}
$$

In particular, the case for disjunctions $\varphi_1 \vee \varphi_2$ introduces a fresh variable $h_l \in H$ (indexed by the position l of the disjunction) that controls which branch is taken. Input variables $v_i \in V_n$ are ignored in the updates.

The input invariants $InputInv(\Delta(q_i), i)$ are similarly defined recursively, and include the same auxiliary variables $h_l \in H$, but ensure input consistency:

$$InputInv(\varphi_1 \wedge \varphi_2, l) = InputInv(\varphi_1, l) \wedge InputInv(\varphi_2, l)$$
$$InputInv(\varphi_1 \vee \varphi_2, l) = \big(h_l \rightarrow InputInv(\varphi_1, l.1)\big) \wedge \big(\neg h_l \rightarrow InputInv(\varphi_2, l.2)\big)$$
$$InputInv(v_i, l) = v_i, \quad InputInv(\neg v_i, l) = \neg v_i, \quad InputInv(q_i, l) = \texttt{true}, \quad InputInv(\phi, l) = \phi\,.$$

9 IMPLEMENTATION AND EXPERIMENTS

We have implemented our method for deciding conjunctive AC and SL formulae as a solver called Sloth (**S**tring **LO**gic **TH**eory solver), extending the Princess SMT solver [Rümmer 2008]. The solver Sloth can be obtained from https://github.com/uuverifiers/sloth/wiki. Hence, Princess provides us with infrastructure such as an implementation of DPLL(T) or facilities for reading input formulae in the SMT-LIBv2 format [Barrett et al. 2010]. Like Princess, Sloth was implemented in Scala. We present results from several settings of our tool featuring different optimizations.

- Sloth-1 The basic version of Sloth, denoted as Sloth-1 below, uses the direct translation of the AFA emptiness problem to checking reachability in transition systems described in Section 8.1. Then, it employs the nuXmv model checker [Cavada et al. 2014] to solve the reachability problem via the IC3 algorithm [Bradley 2012], based on property-directed state space approximation. Further, we have implemented five optimizations/variants of the basic solver: four of them are described below, the last one at the end of the section.
- Sloth-2 Our first optimization, implemented in Sloth-2, is rather simple: We assume working with strings over an alphabet Σ and look for equations of the form $x = a_0 \circ y_1 \circ a_1 \ldots \circ y_n \circ a_n$ where $n \geq 1$, $\forall 0 \leq i \leq n : a_i \in \Sigma^*$ (i.e., a_i are constant strings), and, for every $1 \leq j \leq n$, y_j is a free string variable not used in any other constraint. The optimization replaces such constraints by a regular constraint $(a_0 \circ \Sigma^* \circ a_1 \ldots \circ \Sigma^* \circ a_n)(x)$. This step allows us to avoid many split operations. The optimization is motivated by a frequent appearance of constraints of the given kind in some of the considered benchmarks. As shown by our experimental results below, the optimization yields very significant savings in practice, despite of its simplicity.
- Sloth-3 Our second optimization, implemented in Sloth-3, replaces the use of nuXmv and IC3 in Sloth-2 by our own, rather simple model checker working directly on the generated AFA. In particular, our model checker is used whenever no split operation is needed after the preprocessing proposed in our first optimization. It works explicitly with sets of conjunctive state formulae representing the configurations reached. The initial formula and transition formulae are first converted to DNF using the Tseytin procedure. Then a SAT solver—in particular, sat4j [Berre and Parrain 2010]—is used to generate new reachable configurations and to check the final condition. Our experimental results show that using this simple model checking approach can win over the advanced IC3 algorithm on formulae without splitting.
- Sloth-4 Our further optimization, Sloth-4, optimizes Sloth-3 by employing the intensionally minimal successor computation of Section 8.2 within the IC3-based model checking of nuXmv.
- Sloth-5 Finally, Sloth-5 modifies Sloth-4 by replacing the use of nuXmv with the property directed reachability (i.e., IC3) implementation in the ABC tool [Brayton and Mishchenko 2010].

We present data on two benchmark groups (each consisting of two benchmark sets) that demonstrate two points. First, the main strength of our tool is shown on solving complex combinations of transducer and concatenation constraints (generated from program code similar to that of Example 1.1) that are beyond capabilities of any other solver. Second, we show that our tool is competitive also on simpler examples that can be handled by other tools (smaller constraints with less intertwined and general combinations of rational and concatenation constraints). All the benchmarks fall within the decidable straight-line fragment (possibly extended with the restricted

length constraints). All experiments were executed on a computer with Intel Xeon E5-2630v2 CPU @ 2.60 GHz and 32 GiB RAM.

Complex combinations of concatenation and rational constraints. The first set of our benchmarks consisted of 10 formulae (5 sat and 5 unsat) derived manually from the PHP programs available from the web page of the Stranger tool [Yu et al. 2010]. The property checked was absence of the vulnerability pattern `.*<script.*` in the output of the programs. The formulae contain 7–42 variables (average 21) and 7–38 atomic constraints (average 18). Apart from the Boolean connectives $\wedge$ and $\vee$, they use regular constraints, concatenation, the `str.replaceall` operation, and several special-purpose transducers encoding various PHP functions used in the programs (e.g., `addslashes`, `trim`, etc.).

Results of running the different versions of Sloth on the formulae are shown in Table 1. Apart from the Sloth version used, the different columns show numbers of solved sat/unsat formulae (together with the time used), numbers of out-of-memory runs ("mo"), as well as numbers of sat/unsat instances for which the particular Sloth version provided the best result ("win +/-"). We can see that Sloth was able to solve 9 out of the 10 formulae, and that each of its versions—apart from Sloth-4—provided the best result in at least some case.

Table 1. PHP benchmarks from the web of Stranger.

Program	#sat (sec)	#unsat (sec)	#mo	#win +/-
Sloth-1	4 (178)	5 (6,989)	1	1/0
Sloth-2	4 (83)	5 (5,478)	1	0/2
Sloth-3	4 (72)	5 (3,673)	1	1/2
Sloth-4	4 (93)	4 (6,168)	2	0/0
Sloth-5	4 (324)	4 (4,409)	2	2/1

Our second benchmark consists of 8 challenging formulae taken from the paper [Kern 2014] providing an overview of XSS vulnerabilities in JavaScript programs (including the motivating example from the introduction).

The formulae contain 9–12 variables (average 9.75) and 9–13 atomic constraints (average 10.5). Apart from conjunctions, they use regular constraints, concatenation, `str.replaceall`, and again several special-purpose transducers encoding various JavaScript functions (e.g., `htmlescape`, `escapeString`, etc.). The results of our experiments are shown in Table 2. The meaning of the columns is the same as in Table 1 except that we drop the out-of-memory column since Sloth could handle all the formulae—which we consider to be an excellent result.

Table 2. Benchmarks from [Kern 2014].

Solver	#sat (sec)	#unsat (sec)	#win +/-
Sloth-1	4 (458)	4 (583)	0/2
Sloth-2	4 (483)	4 (585)	0/1
Sloth-3	4 (508)	4 (907)	2/1
Sloth-4	4 (445)	4 (1,024)	1/0
Sloth-5	4 (568)	4 (824)	1/0

These results are the highlight of our experiments, taking into account that we are not aware of any other tool capable of handling the logic fragment used in the formulae.[9]

A Comparison with other tools on simpler benchmarks. Our next benchmark consisted of 3,392 formulae provided to us by the authors of the Stranger tool. These formulae were derived by Stranger from real web applications analyzed for security; to enable other tools to handle the benchmarks, in the benchmarks the `str.replaceall` operation was approximated by `str.replace`.

[9]We tried to replace the special-purpose transducers by a sequence of `str.replaceall` operations in order to match the syntactic fragment of the S3P solver [Trinh et al. 2016]. However, neither Sloth nor S3P could handle the modified formulae. We have not experimented with other semi-decision procedures, such as those implemented within Stranger or SLOG [Wang et al. 2016], since they are indeed a different kind of tool, and, moreover, often are not able to process input in the SMT-LIBv2 format, which would complicate the experiments.

Apart from the ∧ and ∨ connectives, the formulae use regular constraints, concatenation, and the `str.replace` operation. They contain 1–211 string variables (on average 6.5) and 1–182 atomic formulae (on average 5.8). Importantly, the use of concatenation is much less intertwined with `str.replace` than it is with rational constraints in benchmarks from Tables 1 and 2 (only about 120 from the 3,392 examples contain `str.replace`). Results of experiments on this benchmark are shown in Table 3. In the table, we compare the different versions of our Sloth, the S3P solver, and the CVC4 string solver [Liang et al. 2014].[10] The meaning of the columns is the same as in the previous tables, except that we now specify both the number of time-outs (for a time-out of 5 minutes) and out-of-memory runs ("to/mo").

From the results, we can see that CVC4 is winning, but (1) unlike Sloth, it is a semi-decision procedure only, and (2) the formulae of this benchmark are much simpler than in the previous benchmarks (from the point of view of the operations used), and hence the power of Sloth cannot really manifest.

Table 3. Benchmarks from Stranger with `str.replace`.

Solver	#sat (sec)	#unsat (sec)	#to/mo	#win +/-
Sloth-1	1,200 (19,133)	2,079 (3,276)	105/8	30/43
Sloth-2	1,211 (13,120)	2,079 (3,338)	97/5	19/0
Sloth-3	1,290 (6,619)	2,082 (1,012)	14/6	263/592
Sloth-4	1,288 (6,240)	2,082 (1,030)	17/5	230/327
Sloth-5	1,291 (6,460)	2,082 (953)	14/5	768/1,120
CVC4	1,297 (857)	2,082 (265)	13/0	–
S3P	1,291 (171)	2,078 (56)	13/0	–

Despite that, our solver succeeds in almost the same number of examples as CVC4, and it is reasonably efficient. Moreover, a closer analysis of the results reveals that our solver won in 16 sat and 3 unsat instances. Compared with S3P, Sloth won in 22 sat and 4 unsat instances (plus S3P provided 8 unknown and 1 wrong answer and also crashed once). This shows that Sloth can compete with semi-decision procedures at least in some cases even on a still quite simple fragment of the logic it supports.

Our final set of benchmarks is obtained from the third one by filtering out the 120 examples containing `str.replace` and replacing the `str.replace` operations by `str.replaceall`, which reflects the real semantics of the original programs. This makes the benchmarks more challenging, although they are still simple compared to those of Tables 1 and 2. The results are shown in Table 4. The meaning of the columns is the same as in the previous tables. We compare the different versions of Sloth against S3P only since CVC4 does not support `str.replaceall`. On the examples, S3P crashed 6 times and provided 6 times the unknown result and 13 times a wrong result. Overall, although Sloth is still slower, it is more reliable than S3P (roughly 10 % of wrong and 10 % of inconclusive results for S3P versus 0 % of wrong and 5 % of inconclusive results for Sloth).

Table 4. Benchmarks from Stranger with `str.replaceall`.

Program	#sat (sec)	#unsat (sec)	#to/mo	#win +/-
Sloth-1	101 (1,404)	13 (18)	6/0	9/1
Sloth-2	104 (1,178)	13 (18)	3/0	8/5
Sloth-3	103 (772)	13 (19)	4/0	10/1
Sloth-4	101 (316)	13 (23)	6/0	24/2
Sloth-5	102 (520)	13 (20)	5/0	52/4
S3P	86 (11)	6 (26)	0/5	–

As a final remark, we note that, apart from experimenting with the Sloth-1–5 versions, we also tried a version obtained from Sloth-3 by replacing the intensionally minimal successor computation of Section 8.2 by the deterministic successor computation of Section 8.3. On the given benchmark, this version provided 3 times the best result. This underlines the fact that all of the described optimizations can be useful in some cases.

[10] The S3P solver and CVC4 solvers are taken as two representatives of semi-decision procedures for the given fragment with input from SMT-LIBv2.

10 CONCLUSIONS

We have presented the first practical algorithm for solving string constraints with concatenation, general transduction, and regular constraints; the algorithm is at the same time a decision procedure for the acyclic fragment AC of intersection of rational relations of [Barceló et al. 2013] and the straight-line fragment SL of [Lin and Barceló 2016]. The algorithm uses novel ideas including alternating finite automata as symbolic representations and the use of fast model checkers like IC3 [Bradley 2012] for solving emptiness of alternating automata. In initial experiments, our solver has shown to compare favourably with existing string solvers, both in terms of expressiveness and performance. More importantly, our solver can solve benchmarking examples that cannot be handled by existing solvers.

There are several avenues planned for future work, including more general integration of length constraints and support for practically relevant operations like splitting at delimiters and `indexOf`. Extending our approach to incorporate a more general class of length constraints (e.g. Presburger-expressible constraints) seems to be rather challenging since this possibly would require us to extend our notion of alternating finite automata with *monotonic counters* (see [Lin and Barceló 2016]), which (among others) introduces new problems on how to solve language emptiness.

ACKNOWLEDGMENTS

Holík and Janků were supported by the Czech Science Foundation (project 16-24707Y). Holík, Janků, and Vojnar were supported by the internal BUT grant agency (project FIT-S-17-4014) and the IT4IXS: IT4Innovations Excellence in Science (project LQ1602). Lin was supported by European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (Grant Agreement no 759969). Rümmer was supported by the Swedish Research Council under grant 2014-5484.

REFERENCES

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Lukás Holík, Ahmed Rezine, Philipp Rümmer, and Jari Stenman. 2014. String Constraints for Verification. In *CAV*. 150–166.

Davide Balzarotti, Marco Cova, Viktoria Felmetsger, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. 2008. Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. In *S&P*. 387–401.

Pablo Barceló, Diego Figueira, and Leonid Libkin. 2013. Graph Logics with Rational Relations. *Logical Methods in Computer Science* 9, 3 (2013). DOI: http://dx.doi.org/10.2168/LMCS-9(3:1)2013

Pablo Barceló, Leonid Libkin, A. W. Lin, and Peter T. Wood. 2012. Expressive Languages for Path Queries over Graph-Structured Data. *ACM Trans. Database Syst.* 37, 4 (2012), 31.

Clark Barrett, Aaron Stump, and Cesare Tinelli. 2010. The SMT-LIB Standard: Version 2.0. In *Proc. of SMT'10*.

Clark W. Barrett, Cesare Tinelli, Morgan Deters, Tianyi Liang, Andrew Reynolds, and Nestan Tsiskaridze. 2016. Efficient solving of string constraints for security analysis. In *Proceedings of the Symposium and Bootcamp on the Science of Security, Pittsburgh, PA, USA, April 19-21, 2016*. 4–6. DOI: http://dx.doi.org/10.1145/2898375.2898393

Daniel Le Berre and Anne Parrain. 2010. The Sat4j library, release 2.2. *JSAT* 7, 2-3 (2010), 59–6. http://jsat.ewi.tudelft.nl/content/volume7/JSAT7_4_LeBerre.pdf

Jean Berstel. 1979. *Transductions and Context-Free Languages*. Teubner-Verlag.

Armin Biere, Keijo Heljanko, and Siert Wieringa. 2017. AIGER 1.9 and Beyond (Draft). http://fmv.jku.at/hwmcc11/beyond1.pdf (cited in 2017). (2017).

Nikolaj Bjørner, Nikolai Tillmann, and Andrei Voronkov. 2009. Path feasibility analysis for string-manipulating programs. In *TACAS*. 307–321.

Aaron R. Bradley. 2012. Understanding IC3. In *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*. 1–14. DOI: http://dx.doi.org/10.1007/978-3-642-31612-8_1

Robert Brayton and Alan Mishchenko. 2010. ABC: An Academic Industrial-Strength Verification Tool. In *Computer Aided Verification: 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, Tayssir Touili, Byron Cook, and Paul Jackson (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 24–40. DOI: http://dx.doi.org/10.1007/978-3-642-14295-6_5

Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. 2008. EXE: Automatically Generating Inputs of Death. *ACM Trans. Inf. Syst. Secur.* 12, 2 (2008), 10:1–10:38. DOI: http://dx.doi.org/10.1145/1455518.1455522

Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S. Pasareanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. 2011. Symbolic execution for software testing in practice: preliminary assessment. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*. 1066–1071. DOI: http://dx.doi.org/10.1145/1985793.1985995

Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. 2014. The nuXmv Symbolic Model Checker. In *CAV'14 (Lecture Notes in Computer Science)*, Vol. 8559. Springer, 334–342.

Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. 1999. *Model Checking*. The MIT Press, Cambridge, Massachusetts.

Google co. 2015. Google Closure Library (referred in Nov 2015). https://developers.google.com/closure/library/. (2015).

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, Third Edition* (3rd ed.). The MIT Press.

Arlen Cox and Jason Leasure. 2017. Model Checking Regular Language Constraints. *CoRR* abs/1708.09073 (2017). arXiv:1708.09073 http://arxiv.org/abs/1708.09073

Loris D'Antoni, Zachary Kincaid, and Fang Wang. 2016. A Symbolic Decision Procedure for Symbolic Alternating Finite Automata. *CoRR* abs/1610.01722 (2016). http://arxiv.org/abs/1610.01722

Loris D'Antoni and Margus Veanes. 2013. Static Analysis of String Encoders and Decoders. In *VMCAI*. 209–228.

Leonardo De Moura and Nikolaj Bjørner. 2011. Satisfiability modulo theories: introduction and applications. *Commun. ACM* 54, 9 (2011), 69–77.

Volker Diekert. 2002. Makanin's Algorithm. In *Algebraic Combinatorics on Words*, M. Lothaire (Ed.). Encyclopedia of Mathematics and its Applications, Vol. 90. Cambridge University Press, Chapter 12, 387–442.

Laurent Doyen and Jean-François Raskin. 2010. Antichain Algorithms for Finite Automata. In *TACAS'10 (Lecture Notes in Computer Science)*, Vol. 6015. Springer, 2–22. DOI: http://dx.doi.org/10.1007/978-3-642-12002-2_2

Alain Finkel. 1987. A Generalization of the Procedure of Karp and Miller to Well Structured Transition Systems. In *Automata, Languages and Programming, 14th International Colloquium, ICALP87, Karlsruhe, Germany, July 13-17, 1987, Proceedings (Lecture Notes in Computer Science)*, Thomas Ottmann (Ed.), Vol. 267. Springer, 499–508. DOI: http://dx.doi.org/10.1007/3-540-18088-5_43

Xiang Fu and Chung-Chih Li. 2010. Modeling Regular Replacement for String Constraint Solving. In *NFM*. 67–76.

Xiang Fu, Michael C. Powell, Michael Bantegui, and Chung-Chih Li. 2013. Simple linear string constraints. *Formal Asp. Comput.* 25, 6 (2013), 847–891.

Vijay Ganesh, Mia Minnes, Armando Solar-Lezama, and Martin Rinard. 2013. Word equations with length constraints: what's decidable? In *Hardware and Software: Verification and Testing*. Springer, 209–226.

Graeme Gange, Jorge A. Navas, Peter J. Stuckey, Harald Søndergaard, and Peter Schachte. 2013. Unbounded Model-Checking with Interpolation for Regular Language Constraints. In *TACAS'2013 (Lecture Notes in Computer Science)*, Vol. 7795. Springer, 277–291.

Seymour Ginsburg and Edwin H. Spanier. 1966. Semigroups, Presburger formulas, and languages. *Pacific J. Math.* 16, 2 (1966), 285–296. http://projecteuclid.org/euclid.pjm/1102994974

Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: directed automated random testing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*. 213–223. DOI: http://dx.doi.org/10.1145/1065010.1065036

Claudio Gutiérrez. 1998. Solving Equations in Strings: On Makanin's Algorithm. In *LATIN*. 358–373.

Mario Heiderich, Jörg Schwenk, Tilman Frosch, Jonas Magazinius, and Edward Z. Yang. 2013. mXSS attacks: attacking well-secured web-applications by using innerHTML mutations. In *CCS*. 777–788.

Pieter Hooimeijer, Benjamin Livshits, David Molnar, Prateek Saxena, and Margus Veanes. 2011. Fast and Precise Sanitizer Analysis with BEK. In *USENIX Security Symposium*. http://static.usenix.org/events/sec11/tech/full_papers/Hooimeijer.pdf

Pieter Hooimeijer and Westley Weimer. 2012. StrSolve: solving string constraints lazily. *Autom. Softw. Eng.* 19, 4 (2012), 531–559.

Artur Jez. 2016. Recompression: A Simple and Powerful Technique for Word Equations. *J. ACM* 63, 1 (2016), 4:1–4:51. DOI: http://dx.doi.org/10.1145/2743014

Scott Kausler and Elena Sherman. 2014. Evaluation of String Constraint Solvers in the Context of Symbolic Execution. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE '14)*. ACM, New York, NY, USA, 259–270. DOI: http://dx.doi.org/10.1145/2642937.2643003

Christoph Kern. 2014. Securing the Tangled Web. *Commun. ACM* 57, 9 (Sept. 2014), 38–47.

Adam Kiezun and others. 2012. HAMPI: A solver for word equations over strings, regular expressions, and context-free grammars. *ACM Trans. Softw. Eng. Methodol.* 21, 4 (2012), 25.

Nils Klarlund, Anders Møller, and Michael I. Schwartzbach. 2002. MONA Implementation Secrets. *International Journal of Foundations of Computer Science* 13, 4 (2002), 571–586.

Johannes Kloos, Rupak Majumdar, Filip Niksic, and Ruzica Piskac. 2013. Incremental, Inductive Coverability. In *CAV'13 (Lecture Notes in Computer Science)*, Vol. 8044. Springer, 158–173.

Daniel Kroening and Ofer Strichman. 2008. *Decision Procedures.* Springer.

Tianyi Liang, Andrew Reynolds, Cesare Tinelli, Clark Barrett, and Morgan Deters. 2014. A DPLL(T) Theory Solver for a Theory of Strings and Regular Expressions. In *CAV*. 646–662.

Tianyi Liang, Andrew Reynolds, Nestan Tsiskaridze, Cesare Tinelli, Clark Barrett, and Morgan Deters. 2016. An efficient SMT solver for string constraints. *Formal Methods in System Design* 48, 3 (2016), 206–234. DOI: http://dx.doi.org/10.1007/s10703-016-0247-6

Tianyi Liang, Nestan Tsiskaridze, Andrew Reynolds, Cesare Tinelli, and Clark Barrett. 2015. A Decision Procedure for Regular Membership and Length Constraints over Unbounded Strings. In *Frontiers of Combining Systems - 10th International Symposium, FroCoS 2015, Wroclaw, Poland, September 21-24, 2015. Proceedings.* 135–150. DOI: http://dx.doi.org/10.1007/978-3-319-24246-0_9

Anthony Widjaja Lin and Pablo Barceló. 2016. String solving with word equations and transducers: towards a logic for analysing mutation XSS. In *POPL*. 123–136. DOI: http://dx.doi.org/10.1145/2837614.2837641

Blake Loring, Duncan Mitchell, and Johannes Kinder. 2017. ExpoSE: Practical Symbolic Execution of Standalone JavaScript. In *SPIN*.

Gennady S Makanin. 1977. The problem of solvability of equations in a free semigroup. *Sbornik: Mathematics* 32, 2 (1977), 129–198.

John McCarthy. 1980. Circumscription - A Form of Non-Monotonic Reasoning. *Artif. Intell.* 13, 1-2 (1980), 27–39. DOI: http://dx.doi.org/10.1016/0004-3702(80)90011-9

Kenneth L. McMillan. 2003. Interpolation and SAT-Based Model Checking. In *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings.* 1–13. DOI: http://dx.doi.org/10.1007/978-3-540-45069-6_1

Christophe Morvan. 2000. On Rational Graphs. In *FoSSaCS*. 252–266.

Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. 2004. Abstract DPLL and Abstract DPLL Modulo Theories. In *LPAR'04 (LNCS)*, Vol. 3452. Springer, 36–50.

OWASP. 2013. https://www.owasp.org/images/f/f8/OWASP_Top_10_-_2013.pdf. (2013).

Wojciech Plandowski. 2004. Satisfiability of word equations with constants is in PSPACE. *J. ACM* 51, 3 (2004), 483–496.

Wojciech Plandowski. 2006. An efficient algorithm for solving word equations. In *STOC*. 467–476.

Gideon Redelinghuys, Willem Visser, and Jaco Geldenhuys. 2012. Symbolic execution of programs with strings. In *SAICSIT*. 139–148.

Philipp Rümmer. 2008. A Constraint Sequent Calculus for First-Order Logic with Linear Integer Arithmetic. In *Proceedings, 15th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LNCS)*, Vol. 5330. Springer, 274–289.

Jacques Sakarovitch. 2009. *Elements of automata theory.* Cambridge University Press.

Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. 2010. A Symbolic Execution Framework for JavaScript. In *S&P*. 513–528.

Koushik Sen, Swaroop Kalasapur, Tasneem G. Brutch, and Simon Gibbs. 2013. Jalangi: a selective record-replay and dynamic analysis framework for JavaScript. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013.* 488–498. DOI: http://dx.doi.org/10.1145/2491411.2491447

Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. 2000. Checking Safety Properties Using Induction and a SAT-Solver. In *FMCAD (LNCS)*, Vol. 1954. Springer, 108–125.

Deian Tabakov and Moshe Y. Vardi. 2005. Experimental Evaluation of Classical Automata Constructions. In *Logic for Programming, Artificial Intelligence, and Reasoning, 12th International Conference, LPAR 2005, Montego Bay, Jamaica, December 2-6, 2005, Proceedings (Lecture Notes in Computer Science)*, Geoff Sutcliffe and Andrei Voronkov (Eds.), Vol. 3835. Springer, 396–411. DOI: http://dx.doi.org/10.1007/11591191_28

Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. 2014. S3: A Symbolic String Solver for Vulnerability Detection in Web Applications. In *CCS*. 1232–1243.

Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. 2016. Progressive Reasoning over Recursively-Defined Strings. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I.* 218–240. DOI: http://dx.doi.org/10.1007/978-3-319-41528-4_12

Moshe Y. Vardi. 1995. An Automata-Theoretic Approach to Linear Temporal Logic. In *Logics for Concurrency - Structure versus Automata (8th Banff Higher Order Workshop, August 27 - September 3, 1995, Proceedings).* 238–266. DOI: http://dx.doi.org/10.1007/3-540-60915-6_6

Margus Veanes, Pieter Hooimeijer, Benjamin Livshits, David Molnar, and Nikolaj Bjørner. 2012. Symbolic finite state transducers: algorithms and applications. In *POPL*. 137–150.

Hung-En Wang, Tzung-Lin Tsai, Chun-Han Lin, Fang Yu, and Jie-Hong R. Jiang. 2016. String Analysis via Automata Manipulation with Logic Circuit Representation. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I (Lecture Notes in Computer Science)*, Vol. 9779. Springer, 241–260. DOI: http://dx.doi.org/10.1007/978-3-319-41528-4

Gary Wassermann, Dachuan Yu, Ajay Chander, Dinakar Dhurjati, Hiroshi Inamura, and Zhendong Su. 2008. Dynamic test input generation for web applications. In *ISSTA*. 249–260.

Joel Weinberger, Prateek Saxena, Devdatta Akhawe, Matthew Finifter, Eui Chul Richard Shin, and Dawn Song. 2011. A Systematic Analysis of XSS Sanitization in Web Application Frameworks. In *ESORICS*. 150–171.

Fang Yu, Muath Alkhalaf, and Tevfik Bultan. 2010. Stranger: An Automata-Based String Analysis Tool for PHP. In *TACAS*. 154–157. Benchmark can be found at http://www.cs.ucsb.edu/~vlab/stranger/.

Fang Yu, Muath Alkhalaf, Tevfik Bultan, and Oscar H. Ibarra. 2014. Automata-based symbolic string analysis for vulnerability detection. *Formal Methods in System Design* 44, 1 (2014), 44–70.

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

Fang Yu, Tevfik Bultan, and Oscar H. Ibarra. 2011. Relational String Verification Using Multi-Track Automata. *Int. J. Found. Comput. Sci.* 22, 8 (2011), 1909–1924.

Yunhui Zheng, Xiangyu Zhang, and Vijay Ganesh. 2013. Z3-str: a Z3-based string solver for web application analysis. In *ESEC/SIGSOFT FSE*. 114–124.