BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

AUTOMATA IN DECISION PROCEDURES AND FORMAL VERIFICATION

AUTOMATY V ROZHODOVACÍCH PROCEDURÁCH A FORMÁLNÍ VERIFIKACI

PHD THESIS
DISERTAČNÍ PRÁCE

AUTHOR **Ing. PETR JANKŮ**
AUTOR PRÁCE

SUPERVISOR **doc. Mgr. LUKÁŠ HOLÍK, Ph.D.**
ŠKOLITEL

CO-SUPERVISOR **prof. Ing. TOMÁŠ VOJNAR, Ph.D.**
ŠKOLITEL SPECIALISTA

BRNO 2023

Abstract

In this thesis, we propose a fast reduction of the satisfiability of formulae in the straight-line and acyclic fragments to the emptiness problem of alternating finite-state automata (AFA), which is polynomial in most cases. This reduction, in combination with advanced model checking algorithms such as IC3, provides the first practical algorithm for solving string constraints involving concatenation, finite-state transducers and regular constraints. Furthermore, we introduce a new fragment of string constraints called chain-free and its relaxation called weakly chaninng, along with decision procedures for these fragments. It is important to mention that these new fragments generalize both the straight-line fragment and the acyclic form. Additionally, we presented a method for checking the satisfiability of string constraints, in particular with string-to-number conversion, using parametric flat automata (PFA). This procedure is complemented by an algorithm for converting string constraints to linear formulas in polynomial time with a search space bounded by PFA. In conclusion, we propose and integrate an improved Parikh abstraction into the string solver Sloth for solving length constraints.

Abstrakt

V této práci navrhneme rychlou redukci splnitelnosti formulí v straight-line a acyklickém fragmentu na problém prázdnosti alternujících konečných automatů (AFA), která je ve většině případů polynomiální. Tato redukce v kombinaci s pokročilými algoritmy pro kontrolu modelů, jako je IC3, poskytuje první praktický algoritmus pro řešení omezení nad řetězci zahrnujících konkatenaci, převodníky a regulární omezení. Dále zavedeme nový fragment řetězcových omezení zvaný chain-free a jeho relaxaci zvanou weakly chaninng spolu s rozhodovacími procedurami pro tyto fragmenty. Je důležité zmínit, že tyto nové fragmenty zobecňují jak straight-line fragment, tak acyklickou formu. Navíc představíme metodu pro ověření splnitelnosti omezení nad řetězci, zejména s převodem mezi řetězci a čísly, pomocí parametrických plochých automatů (PFA). Tento postup je doplněn o algoritmus pro převod omezení nad řetězci na lineární formule v polynomiálním čase s prohledávacím prostorem ohraničeným PFA. Na závěr navrhneme vylepšenou Parikhovu abstrakci pro řešení délkových omezení pro straight-line fragment.

Keywords

String solving, alternating finite automata, decision procedure, IC3, satisfiability modulo theories, program verification, string constraints, automata, Parikh image.

Klíčová slova

Řešení řetězců, střídavé konečné automaty, rozhodovací procedura, IC3, splnitelnost modulo teorie (SMT), verifikace programů, omezení nad řetězci, automaty, Parikhův obraz.

Reference

JANKŮ, Petr. *Automata in Decision Procedures and Formal Verification.* Brno, 2023. PhD thesis. Brno University of Technology, Faculty of Information Technology. Supervisors doc. Mgr. Lukáš Holík, Ph.D., prof. Ing. Tomáš Vojnar, Ph.D.

Rozšířený abstrakt

Řetězce, jejž jsou základem moderních programovacích jazyků, hrají klíčovou roli při reprezentaci a manipulaci s textovými daty, zejména ve webových aplikacích. Jejich přizpůsobivost umožňuje interakci v mateřských jazycích uživatelů a usnadňuje komunikaci mezi systémy, což ilustrují formáty XML a JSON. Nicméně, manipulace s těmito řetězci, zejména při zpracování nedůvěryhodných uživatelských dat, může vést k vážným bezpečnostním zranitelnostem, jako je Cross-Site Scripting (XSS) a SQL injection. Navzdory zvyšující se informovanosti jsou tyto zranitelnosti stále rozšířené. Základní obranou vůči těmto zranitelnostem je sanitizace nedůvěryhodných dat pomocí specifických metod. Ukázalo se však, že tyto metody nemusí být vždy správně použity. Pro zajištění odolnosti proti těmto rizikům je proto nezbytná verifikace programu, ať už dynamická, nebo statická analýza. Dynamická analýza sice poskytuje posouzení v reálném čase, ale často trpí problémem "nízkého pokrytí kódu". Proto se standardně používá statická analýza. Mezi oblíbené techniky statické analýzy pro analyzování řetězců patří symbolická exekuce, které ve svém jádru používají řešiče omezení nad doménou řetězců, tzv. string solvery. Tyto řešiče obvykle analyzují řetězcová omezení, která kombinují relační omezení reprezentovaná převodníky, slovními rovnice, omezeními délky řetězce a konverzi mezi řetězci a čísly. Ačkoli je teorie nad řetězcových omezení obecně nerozhodnutelná, byly nalezeny smysluplné a expresivní podtřídy řetězcových logik, pro které je problém splnitelnosti rozhodnutelný. Mezi takové významné třídy patří acyklický fragment a straight-line fragment.

V této práci poskytujeme první praktický řešič řetězcových omezení, který dokáže analyzovat omezení zahrnujících konkatenaci, konečnou transdukci a náležitost v regulárním jazyku. Navíc je pro tento řešič garantována úplnost a terminace pro formule v straight-line a acyklickém fragmentu. Hlavní výzvou je omezující složitost teorie řetězců v nejhorším případě (dvojnásobný exponenciální čas), která je exponenciálně těžší než teorie bez konečněstavových transdukcí. Navrhujeme proto metodu, která využívá kompaktní alternující konečné automaty jako kompaktní symbolické reprezentace řetězcových omezení. Na rozdíl od předchozích přístupů využívajících nedeterministické automaty nabízí alternace nejen exponenciální úsporu místa při reprezentaci booleovských kombinací převodníků, ale také možnost stručné reprezentace jinak nákladných kombinací převodníků a konkatenace. Odůvodnění prázdnosti jazyka AFA vyžaduje průzkum stavového prostoru v grafu exponenciální velikosti, k čemuž se používají algoritmy pro kontrolu modelu (např. IC3). Náš algoritmus prokázal efektivnost na benchmarcích, které jsou odvozeny z analýzy webových aplikací a dalších příkladů v literatuře.

Následně jsme navrhli nový rozhodnutelný fragment řetězcových omezení, tzv. slabě řetězcová omezení, pro který ukazujeme, že problém splnitelnosti je rozhodnutelný. Tento fragment posouvá hranice rozhodnutelnosti řetězcových omezení tím, že zobecňuje stávající straight-line i acyklický fragment řetězcové logiky. Vyvinuli jsme prototypovou implementaci naší nové rozhodovací procedury a začlenili ji do existujícího frameworku, který používá CEGAR s podaproximací řetězcových omezení na základě zploštění. Naše experimentální výsledky ukazují konkurenceschopnost a přesnost nového frameworku.

Dále pak jsme navrhli přístup, který dokáže efektivně podporovat jak konverzi mezi řetězci a čísly, tak další běžné typy řetězcových omezení. Zejména řešení řetězcových omezení s převodem řetězců na čísla je pro nejmodernější řešiče velmi náročné. Náš přístup využívá konceptu parametrických plochých automatů (PFA), které se ukázali být klíčovým nástrojem pro efektivní zpracování těchto řetězcových omezení. Experimentální výsledky ukazují, že náš přístup výrazně překonává nejmodernější řetězcové řešiče na benchmarcích, které zahrnují i konverzi mezi řetězci a čísly.

Na závěr navrhujeme vylepšenou verzi Parikhovy obrazové abstrakce konečných automatů pro řešení omezení nad délkami řetězců. Tuto abstrakci integrujeme do řetězcového řešiče SLOTH, kde kromě řešení délkových omezení využíváme naši abstrakci také ke zrychlení řešení dalších typů omezení. Experimentální výsledky ukazují, že naše rozšíření SLOTH má dobré výsledky jak na jednoduchých tak i na složitých benchmarcích.

Automata in Decision Procedures and Formal Verification

Declaration

Prohlašuji, že jsem tuto disertační práci vypracoval samostatně pod vedením doc. Mgr. Lukáše Holíka, Ph.D. a prof. Ing. Tomáše Vojnara, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

. . . .
Petr Janků
September 30, 2023

Acknowledgements

First of all, I would like to express my gratitude to my supervisor Lukáš Holík for his incredible efforts and patience. I know it has not always been easy with me, but his unwavering faith in me and his encouragement has been a tremendous source of support. Without him, this thesis would never have come to be, and for that I owe him my greatest thanks. I would also like to thank my co-supervisor, Tomáš Vojnar, for his valuable time and all the support he gave me during my studies. I cannot forget to thank my co-authors, especially Lenka Turoňová, Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bui Phi Diep, Anthony W. Lin, Philipp Rümmer, Yu-Fang Chen, Julian Dolby, Hsin-Hung Lin, and Wei-Cheng Wu. Thanks also to all the people in the VeriFIT group, especially Martin Hruška, Tomáš Fiedor, Ondřej Lengál and Lenka Turoňová. It was great to be part of this group. Finally, I would like to express my gratitude to my family and friends for their endless support and especially to my beloved wife Helena, who has been my greatest source of strength and encouragement.

Contents

1 Introduction **2**
- 1.1 Contribution of This Thesis . . . 5

2 Preliminaries **8**

3 String Constraints **9**
- 3.1 String Language . . . 9
- 3.2 Decidability and Complexity of Existing Decision Procedures . . . 10
  - 3.2.1 Acyclic Form . . . 11
  - 3.2.2 Straight-Line Fragment . . . 12

4 Contributions **14**
- 4.1 String Constraints with Concatenation and Transducers Solved Efficiently . . . 14
- 4.2 Chain-Free String Constraints . . . 15
- 4.3 Efficient Handling of String-Number Conversion . . . 18
- 4.4 Solving String Constraints with Approximate Parikh Image . . . 20

5 Conclusions and Future Directions **21**
- 5.1 Summary of the Contributions . . . 21
- 5.2 Further Directions . . . 21

Bibliography **24**

A Papers **34**
- A.1 String constraints with concatenation and transducers solved efficiently . . . 35
- A.2 Chain-Free String Constraints . . . 67
- A.3 Efficient handling of string-number conversion . . . 84
- A.4 Solving String Constraints with Approximate Parikh Image . . . 99

Chapter 1

Introduction

Strings are a fundamental data type in many, if not all, modern programming languages. They are uniquely important because of their versatility and unique ability to effectively represent, process and manipulate textual data. This is particularly crucial in the realm of web applications where they facilitate communication with users in their native language. Modern programming languages, such as JavaScript, Python, Java, and PHP, reflect this essential status of strings by offering an extensive collection of built-in functions designed specifically for working with strings. These functions provide convenient and efficient string manipulation, ranging from basic operations such as concatenation, length and substring, to more complex functions such as match, replace, split and parseInt. In addition, strings have become indispensable in inter-systemand program communication, where they represent values of data types other than strings. This is particularly important when processing or creating textbased XML and JSON file formats, which are heavily utilized in data exchange between servers and web applications, further highlighting the continued and increasing importance of strings in modern software development.

However, string manipulation also comes with significant risk of errors. The extensive use of string operations, particularly for processing untrusted user data, combined with potential built-in string functions, often leads to serious vulnerabilities in web applications. The most prominent of these vulnerabilities are Cross-Site Scripting (XSS) and Injection Flaws, like SQL Injection. XSS attacks, a prevalent issue in modern web applications, occur when untrusted data is passed to other users without proper sanitization, allowing potentially dangerous strings to be interpreted as code by a browser. On the other hand, SQL Injection vulnerabilities arise from the improper usage of user input when constructing database statements. If untrusted data is not adequately sanitized, malicious actors can manipulate queries and gain unauthorized access to the database, for example. Despite increased awareness of these vulnerabilities and efforts to remove them, these vulnerabilities persist on OWASP's list [75, 76, 77] of the most serious web application vulnerabilities over the years. This illustrates that despite increased understanding, these vulnerabilities are still widespread and cause significant damage.

Renowned companies like Google, Facebook, Adobe and Mozilla financially reward anyone who discovers vulnerabilities in their web applications, including security flaws like cross-site scripting (XSS) or SQL injection. For example, Google is offering up to $10,000 as a reward [53]. A less visible but no less serious consequence of these security flaws is the amount of time websites are down. For organizations whose business depends on web technologies, such downtime is a major financial burden. In the event of successful cyber-attacks, especially when XSS is involved, downtime can last for days or weeks until the site

can be made secure again. Simple math shows that if your website generates $150 per hour, downtime due to XSS can cost you between $5,000 and $30,000 if the website is down for two to ten days. Research from 2014 by [90] even showed that even minor SQL injection attacks can have a financial impact of up to $200,000. Another article [95] from the same year published on Ars Technica claims that the US Navy spent more than half a million dollars to address a single SQL injection attack that caused more than 70 people to be unable to continue their transactions for several months. These costs also include the time and resources spent identifying and closing the security gaps that enabled the attack in the first place.

To prevent these vulnerabilities, untrusted data is sanitized using specific functions that escape, i.e., replace potentially dangerous characters with a different sequence of characters, or remove potentially dangerous strings. While modern programming languages provide their own sanitizers, developers may need to create custom sanitizers to meet specific performance or functional constraints. Nonetheless, the correct creation of these sanitizers is a challenging task, as it is known that custom implementations can often contain bugs [50]. In addition, using the same sanitizer multiple times or in different order can inadvertently introduce new vulnerabilities, as demonstrated in Example 1.0.1.

Example 1.0.1. Consider the following JavaScript code snippet adapted from [58, 69]:

```
var x = goog.string.htmlEscape(name);
var y = goog.string.escapeString(x);
nameElem.innerHTML = '<button onclick= "view(\'' + 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 view 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 [35] string functions htmlEscape and escapeString. Here, htmlEscape converts reserved characters in HTML such as &, <, and ' to their respective HTML entity names &, < and '. On the other hand, escapeString will escape certain metacharacters, e.g., the character ' and " are replaced by \' and \". Inputting the value Tom & Jerry into name gives the desired HTML markup:

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

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

```
<button onclick="view('&#39;);script();//')">&#39;);script();//')</button>
```

Before this string is inserted into the DOM via innerHTML, an implicit browser transduction will take place [48, 104], i.e., HTML-unescaping the string inside the onclick attribute and then invoking the attacker's script script() after view. This subtle DOM-based XSS bug is due to calling the right escape functions, but in wrong order. □

It is crucial to address another significant yet often overlooked vulnerability related to string manipulation, known as buffer overflow. These vulnerabilities occur when a program attempts to store more data in a buffer than it can hold, resulting in data overflow into neighboring memory locations and causing data corruption or overwrites. As a consequence, this can lead to crashes, security vulnerabilities, and unpredictable behavior. Attackers could exploit these vulnerabilities to execute arbitrary code on a system, potentially resulting in a system takeover or theft of sensitive data. This issue is especially concerning when a string is copied into a buffer without proper length checking.

Given the significance and inherent risks of string manipulation, particularly in the context of web applications that are especially vulnerable due to their global availability,

it is absolutely essential to perform some form of program verification. Fortunately, there are two main approaches for verifying the security of a program, namely dynamic and static analysis. Dynamic analysis involves testing the application as a whole unit, using a set of specific inputs. However, its main drawback lies in its lack of reliability since certain program paths can only be executed if certain inputs are passed to the program as parameters. Thus, it is highly unlikely that a dynamic analysis could thoroughly test the program with all possible inputs due to its inherent limitations. This becomes even more apparent when we consider web applications. Here, the complexity of the analysis increases significantly as not only the range of input values (the value space) needs to be considered, but also the different sequences of user interactions with the interface (the event space). As a consequence, the number of potential execution paths itself increases, making systematic exploration impractical. This often leads to what is commonly referred to as the "low code coverage" issue in dynamic analysis.

To overcome these limitations, a standard approach is to use static analysis, which aims for good or complete coverage of the program under analysis. However, static analysis introduces its own challenges, such as the existence of false-positive results, which arise due to an over-approximation of the program's behavior. In order to address this issue, a technique called *symbolic execution* [60] can be used. Symbolic execution is a program analysis technique that operates on symbolic inputs instead of concrete values. It evaluates program as functions of these symbolic inputs while maintaining path conditions, which represent the symbolic values along a specific execution path. This concept is further expanded by *dynamic symbolic execution* [91, 25, 26, 27, 45, 92], which gathers symbolic constraints from concrete execution traces and enables the exploration of different execution paths by dynamically monitoring the executed instructions. If a branch condition in one of the extracted symbolic traces is selected and negated, alternative paths can be explored.

In order to determine the feasibility of these modified path conditions, the typical approach involves reducing the problem to the satisfiability of a formula. More precisely, program statements within the path are translated into equivalent constraints in *Static Single Assignment* (SSA) form, which are subsequently solved by a constraint solver. This solver must be able to solve constraints involving different theories or data domains, such as strings, integers and Booleans. These specifications are met by the SMT (Satisfiability Modulo Theories) solver that is commonly used in symbolic executions. By integrating specialized theory solvers, an SMT extends the capabilities of SAT solvers to handle formulas expressed across different theories. This integration is typically achieved within the DPLL(T) framework [73], which is employed by advanced SMT solvers. Within this architecture, an incremental propositional SAT solver initially searches for a truth assignment that satisfies the formula at the propositional level. If successful, this assignment is forwarded to a theory solver, which applies a specific calculus tailored to that theory. The theory solver evaluates whether a saturated configuration is achieved and, based on the results, either confirms the satisfiability of the input formula or provides additional constraints to the SAT solver in the form of conflict clauses or lemmas. This iterative process continues until no conflicts are detected or an irreparable conflict arises.

Driven by the aforementioned importance of strings and their pivotal role in software verification, especially within the symbolic execution model and the utilization of SMT solvers, attention has increasingly shifted towards a specialized category of solvers designed specifically for string manipulation, commonly known as string solvers. Their importance and interest in academic community has grown significantly over the last twenty years, as evidenced by numerous studies [11, 14, 66, 68, 67, 74, 82, 83, 39, 20, 94, 109, 108, 18, 19,

17, 16, 4, 5, 2, 97, 29, 32, 31, 30, 49, 6, 7, 21, 103, 63, 105, 107, 106, 10, 59, 87, 33, 34, 38, 65, 9, 36, 42, 50, 51, 89, 100, 102]. A practical string solver must be able to handle a wide range of string constraints, including basic ones such as word equations, regular expressions, and length constraints. Additionally, it must also be capable of managing various complex constraints, which involve string transformations (e.g., in the form of transducers), `replaceAll`, `substring`, `indexOf` functions, and conversions between integers and strings.

Solving constraints over strings is still a complex and challenging task compared to constraints over integer/real arithmetic, which are well-studied and have already powerful algorithms such as the simplex algorithm. The challenge mainly arises from the diverse range of string operations that can be incorporated into a string theory. Even when considering the theory of strings with only the concatenation operation, existing string solvers are not able to handle this theory in its full generality in a sound and complete manner, despite the existence of a theoretical decision procedure for this problem [40, 47, 56, 70, 79, 80]. Adding an additional operation such as string length comparison further complicates the situation, since in that case decidability is a long-standing open problem [44]. The complexity and limitations of solving string constraints reach a peak when dealing with the full class of string constraints, which includes transducers and string-to-number conversions in addition to concatenation and length constraints. For this full class, it is well known that the satisfiability problem is proven to be undecidable in general [72, 32], even for a simple formula of the form $\mathcal{T}(x, x)$, where $\mathcal{T}$ is a rational transducer and x is a string variable. However, this theoretical barrier has not prevented the development of numerous efficient solvers such as Z3 [39, 20, 94], Z3Str/2/3/4/3RE [109, 108, 18, 19, 17, 16], cvc4/5 [11, 14, 66, 68, 67, 74, 82, 83], S3P [96, 97] and Trau [4, 5, 1, 2, 8]. These tools implement semi-decision algorithms to handle a variety of string constraints. Although these tools are usually sound in the sense that they return the correct answer upon termination, they do not provide completeness guarantees. Another direction of research is to find meaningful and expressive subclasses of string logics for which the satisfiability problem is decidable. Such classes include the acyclic form of Norn [6, 7], the solved form fragment [44], and also the straight-line fragment [29, 32, 31, 30, 69, 49].

1.1 Contribution of This Thesis

In the context of this thesis, which focuses on addressing the problems described above, several papers upon which this thesis is based have been published: [49] published at POPL'18, [8] published at ATVA'19 (Best Paper Award), [2] published at PLDI'20, and [55] published at EUROCAST'19. My contributions to these papers are as follows:

- In the paper [49], I participated in the creation of the decision procedure, the development of the Sloth tool, and the preparation of experiments.
- In the context of paper [8], I focused on developing the chain-free fragment and its theory, developing an experimental tool Trau+, and creating and conducting experiments.
- Regarding publication [1], my involvement included collaborative development of the experimental tool Z3-Trau, creating a validator for checking the results of the used tools, and contributing to the experimental section of the paper.
- In the case of paper [55], I contributed to the writing of the text, software development, and was the author of the main idea of the article.

All the mentioned papers are included to this thesis and can be found in Appendix A. Here is a brief overview of the main contributions of this thesis:

- We proposed a fast reduction of the satisfiability of formulae in the straight-line and acyclic fragment [12] to the emptiness problem of *alternating finite-state automata* (AFA). This reduction can be exponential in the worst case depending on the number of concatenation operations, but otherwise polynomial in the size of the formula. To decide the emptiness of AFA, we combined this reduction with fast model checking algorithms (namely IC3 [22]), which led to the first practical algorithm for handling string constraints with concatenation, finite-state transducers (hence, also `replaceAll`), and regular constraints. Furthermore, we obtained a simpler proof for the decidability and PSPACE-membership of the acyclic fragment of the intersection of rational relations of [12], which was fundamentally used in [69]. Additionally, we have defined optimized translations from AFA emptiness to reachability over Boolean transition systems. The implementation of these translations can be found in the string solver SLOTH [49], whose performance we extensively tested, especially in the context of HTML5 applications. The results show that in many practical cases, the translation to AFAs can circumvent the worst-case EXPSPACE complexity.

- We introduced a new decidable fragment of string constraints, called *chain-free* [8]. This fragment strictly generalizes both the existing straight-line and acyclic fragments [69, 6] and provides a precise characterization of the decidability limitations of general relational/transducer constraints combined with concatenation. Simultaneously, we introduced a relaxation of the chain-free fragment, which is called *weakly chaining*. This fragment allows special chains with length preserving relational constraints. Decision procedures have been developed for these new fragments, focusing on solving the satisfiability problem for both chain-free and weakly chaining constraints. To validate these new procedures, a prototype was created, and experimental results demonstrate the effectiveness and generality of our technique, both based on benchmarks from the literature and new benchmarks.

- An efficient procedure for checking the satisfiability of string constraints, especially with string-to-number conversion, has been proposed using the concept of *parametric flat automata* (PFA). These automata extend the concept of flat automata [4] and have proven to be a key tool for efficiently handling these string constraints. Furthermore, we introduced an algorithm that allows to translates the satisfiability problem of string constraints to the satisfiability problem of a linear formula in polynomial-time, assuming that the search space is restricted by the PFA. In order to demonstrate the efficiency and performance of this approach, an open source tool called Z3-TRAU [1] was developed. Experimental results show that our approach is effective on standard benchmarks as well as on real-world benchmarks.

- The decision procedure contained in the SLOTH string solver for the straight-line fragment faces challenges in efficiently solving arithmetic constraints over strings. This problem stems from the fact that the given procedure utilizes AFA, where the emptiness problem is subsequently solved by model checking. In response to this, we present an extension of the given decision procedure, which offers the possibility of better handling arithmetic constraints over strings. The key feature of this extension is the creation of Parikh images for each AFA and the definition of operations between them, which effectively addresses arithmetic constraints over strings. Although our

extension provides only an approximation of the existing solution, experimental results have shown that it is sufficiently accurate in real-world benchmarks.

Outline. In Chapter 2, we recall the relevant definitions from logic and automata theory. In Chapter 3, we define the basic string constraint language and give a brief overview of decidability and complexity over these constraints. Chapter 4 describes a brief overview of the contributions of this thesis, and finally Chapter 5 concludes and discusses future work. Appendix A then contains the papers that form the main part of this thesis.

Chapter 2

Preliminaries

Sets and strings. We use $\mathbb{N}$, $\mathbb{Z}$ to denote the sets of natural numbers and integers, respectively. A finite set Σ of *letters* is an *alphabet*, a sequence of symbols $a_1 \cdots a_n$ from Σ is a *word* or a *string* over Σ, with its *length* n denoted by $|w|$, ϵ is the *empty word* with $|\epsilon| = 0$, it is a neutral element with respect to string concatenation $\circ$, and Σ^* is the set of all words over Σ including ϵ.

Logic. Given a predicate formula, an occurrence of a predicate is *positive* if it is under an even number of negations. A formula is in *disjunctive normal form* (DNF) if it is a disjunction of *clauses* that are themselves conjunctions of (negated) predicates. We write $\Psi[x/t]$ to denote the formula obtained by substituting in the formula Ψ each occurrence of the variable x by the term t.

(Multi-tape)-Automata and transducers. A *Finite Automaton* (FA) over an alphabet Σ is a tuple $\mathcal{A} = \langle Q, \Delta, I, F \rangle$, where Q is a finite set of *states*, $\Delta \subseteq Q \times \Sigma_\epsilon \times Q$ with $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$ is a set of *transitions*, and $I \subseteq Q$ (resp. $F \subseteq Q$) are the *initial* (resp. *accepting*) states. $\mathcal{A}$ accepts a word w iff there is a sequence $q_0 a_1 q_1 a_2 \cdots a_n q_n$ such that $(q_{i-1}, a_i, q_i) \in \Delta$ for all $1 \leq i \leq n$, $q_0 \in I$, $q_n \in F$, and $w = a_1 \circ \cdots \circ a_n$. The *language* of $\mathcal{A}$, denoted $\mathcal{L}(\mathcal{A})$, is the set all accepted words.

Given $n \in \mathbb{N}$, a *n-tape automaton* $\mathcal{T}$ is an automaton over the alphabet $(\Sigma_\epsilon)^n$. It *recognizes* the relation $\mathcal{R}(\mathcal{T}) \subseteq (\Sigma^*)^n$ that contains vectors of words $(w_1, w_2, \ldots, w_n)$ for which there is $(a_{(1,1)}, a_{(2,1)}, \ldots, a_{(n,1)}) \cdots (a_{(1,m)}, a_{(2,m)}, \ldots, a_{(n,m)}) \in \mathcal{L}(\mathcal{T})$ with $w_i = a_{(i,1)} \circ \cdots \circ a_{(i,m)}$ for all $i \in \{1, \ldots, n\}$. A n-tape automaton $\mathcal{T}$ is said to be *length-preserving* if its transition relation $\Delta \subseteq Q \times \Sigma^n \times Q$. A *transducer* is a 2-tape automaton.

Let us recall some well-know facts about the class of multi-tape automata. First, the class of n-tape automata is closed under union but not under complementation nor intersection. However, the class of *length-preserving* multi-tape automata is closed under intersection. Multi-tape automata are closed under composition. Let $\mathcal{T}$ and $\mathcal{T}'$ be two multi-tape automata of dimension n and m, respectively, and let $i \in \{1, \ldots, n\}$ and $j \in \{1, \ldots, m\}$ be two indices. Then, it is possible to construct a $(n+m-1)$-tape automaton $\mathcal{T} \wedge_{(i,j)} \mathcal{T}'$ which accepts the set of words $(w_1, \ldots, w_n, u_1, \ldots, u_{j-1}, u_{j+1}, \ldots, u_m)$ if and only if $(w_1, \ldots, w_n) \in \mathcal{R}(\mathcal{T})$ and $(u_1, \ldots, u_{j-1}, w_i, u_{j+1}, \ldots, u_m) \in \mathcal{R}(\mathcal{T}')$. Furthermore, we can show that multi-tape automata are closed under permutations: Given a permutation $\sigma : \{1, \ldots, n\} \to \{1, \ldots, n\}$ and a n-tape automaton $\mathcal{T}$, it is possible to construct a n-tape automaton $\sigma(\mathcal{T})$ such that $\mathcal{R}(\sigma(\mathcal{T})) = \{(w_{\sigma(1)}, \ldots, w_{\sigma(n)}) \,|\, (w_1, w_2, \ldots, w_n) \in \mathcal{R}(\mathcal{T})\}$. Finally, given a n-tape automaton $\mathcal{T}$ and a natural number $k \geq n$, we can construct a k-tape automaton s. t. $(w_1, \ldots, w_k) \in \mathcal{R}(\mathcal{T}')$ if and only if $(w_1, \ldots, w_n) \in \mathcal{R}(\mathcal{T})$.

Chapter 3

String Constraints

We start by recalling a general string constraint language, which includes concatenation, finite-state transducers, and regular expression matching. We will then extend this language with additional constraints such as ReplaceAll, IndexOf, and string-number conversion. Subsequently, we will focus on existing decision procedures and their complexity, where we discuss in more detail two important fragments of this language: the acyclic form and the straight-line fragment.

3.1 String Language

The syntax of a string formula Ψ over an alphabet Σ and a set of variables $\mathbb{X}$ is as follows:

$$
\begin{aligned}
\Psi &::= \varphi \mid \Psi \wedge \Psi \mid \Psi \vee \Psi \mid \neg\Psi \\
\varphi &::= \mathcal{A}(t_{str}) \mid R(t_{str}, t_{str}) \mid t_{ar} \geq t_{ar} \\
R &::= \mathcal{T} \mid = \\
t_{str} &::= \epsilon \mid x \mid t_{str} \circ t_{str} \\
t_{ar} &::= k \mid |t_{str}| \mid t_{ar} + t_{ar}
\end{aligned}
$$

It is a Boolean combination of memberships, relational, and arithmetic constraints over string terms t_{str} (i.e., concatenations of variables in $\mathbb{X}$). *Membership constraints* denote membership in the language of a finite-state automaton $\mathcal{A}$ over Σ. *Relational constraints* denote either an equality of string terms, which we normally write as $t = t'$ instead of $=(t, t')$, or that the terms are related by a relation recognised by a transducer $\mathcal{T}$. (Observe that the equality relations can be also expressed using length preserving transducers.) Finally, arithmetic terms t_{ar} are linear functions over term lengths and integers, and arithmetic constraints are inequalities of arithmetic terms. We refer to all previous constraints as *Basic constraints*.

Furthermore, we introduce so-called *Extended constraints*, which, in addition to basic constraints, also include constraints like ReplaceAll, IndexOf, and string-number conversion. The relational constraint ReplaceAll is defined by the function $\mathsf{replaceAll}(x, p, y)$, where x and y are string terms, while p can be either a string term or a regular expression. This function replaces all occurrences of p in x with the expression y. Additionally, the arithmetic constraint IndexOf is described by the function $\mathsf{IndexOf}(x, y)$, where x and y are string terms. This function returns the position of the occurrence of x in y or returns 0 if x is not found in y. Finally, the string-number conversion constraint is defined by the function $\mathsf{toNum}(x)$, where x is a string term. If $x \in [0, 9]^+$, this function returns the number represented by the string x. However, if $x \notin [0, 9]^+$, it returns the value -1.

String formulae allow using negation with one restriction, namely, constraints that are *not invertible* must have only positive occurrences. General transducers are not invertible, it is not possible to negate them. Regular membership, length preserving relations (including equality), and length constraints are invertible.

To simplify presentation, we do not consider *mixed* string terms t_{str} that contain, besides variables of $\mathbb{X}$, also symbols of Σ. This is without loss of generality because a mixed term can be encoded as a conjunction of the pure term over $\mathbb{X}$ obtained by replacing every occurrence of a letter $a \in \Sigma$ by a fresh variable x and the regular membership constraints $\mathcal{A}_a(x)$ with $\mathcal{L}(\mathcal{A}_a) = \{a\}$. Observe also that membership and equality constraints may be expressed using transducers.

Semantics. We describe the semantics of our logic using a mapping η, called *interpretation*, that assigns to each string variable in $\mathbb{X}$ a word in Σ^*. Extended to string terms by $\eta(t_{s_1} \circ t_{s_2}) = \eta(t_{s_1}) \circ \eta(t_{s_2})$. Extended to arithmetic terms by $\eta(|t_s|) = |\eta(t_s)|$, $\eta(k) = k$ and $\eta(t_i + t_i') = \eta(t_i) + \eta(t_i')$. Extended to atomic constraints, η returns a truth value:

$$\begin{aligned} \eta(\mathcal{A}(t_{str})) = \top \quad &\text{iff} \quad \eta(t_{str}) \in \mathcal{L}(\mathcal{A}) \\ \eta(R(t_{str}, t'_{str})) = \top \quad &\text{iff} \quad (\eta(t_{str}), \eta(t'_{str})) \in \mathcal{R}(R) \\ \eta(t_{i_1} \leq t_{i_2}) = \top \quad &\text{iff} \quad \eta(t_{i_1}) \leq \eta(t_{i_2}) \end{aligned}$$

Given two interpretations η_1 and η_2 over two disjoint sets of string variables $\mathbb{X}_1$ and $\mathbb{X}_2$, respectively. We use $\eta_1 \cup \eta_2$ to denote the interpretation over $\mathbb{X}_1 \cup \mathbb{X}_2$ such that $(\eta_1 \cup \eta_2)(x) = \eta_1(x)$ if $x \in \mathbb{X}_1$ and $(\eta_1 \cup \eta_2)(x) = \eta_2(x)$ if $x \in \mathbb{X}_2$.

The truth value of a Boolean combination of formulae under η is defined as usual. If $\eta(\Psi) = \top$ then η is a *solution* of Ψ, written $\eta \models \Psi$. The formula Ψ is *satisfiable* iff it has a solution, otherwise it is *unsatisfiable*.

A relational constraint is said to be *left-sided* if and only if it is on the form $R(x, t_{str})$ where $x \in \mathbb{X}$ is a string variable and t_{str} is a string term. Any string formula can be transformed into a formula where all the relational constraints are left-sided by replacing any relational constraint of the form $R(t_{str}, t'_{str})$ by $R(x, t'_{str}) \wedge x = t$ where x is fresh.

A formula Ψ is said to be *concatenation free* if and only if for every relational constraint $R(t_{str}, t'_{str})$, the string terms t_{str} and t'_{str} appearing in the parameters of any relational constraints in Ψ are variables (i.e., $t_{str}, t'_{str} \in \mathbb{X}$).

3.2 Decidability and Complexity of Existing Decision Procedures

In 1946, Quine [81] presented a proof that the first-order theory of word equations (specifically, word equations with Boolean connectives and quantification over variables) is undecidable. This proof is based on an equivalence with the first-order theory of arithmetic, which was already known to be undecidable. Subsequent efforts were then dedicated to identifying specific subclasses of word equations where decidability would be achievable. A major breakthrough came in 1977 when Makanin [70] in his pioneering work introduced a decision procedure for word equations without quantifiers, i.e., Boolean combinations of equalities and inequalities, where string variables can be assigned words of arbitrary length. Makanin not only proved the decidability of this problem, but also laid the foundation for further advances in the field. Subsequent research has gradually reduced the complexity of Makanin's algorithm to EXPSPACE [54, 88, 62, 46]. Subsequently, in 1999, Plandowski made a breakthrough when he was the first to show that word equations can be solved in

PSPACE due to the word compression technique [78]. Jez, in a series of papers [56, 57], applied a new technique called recompression to word equations. His approach not only simplified the existing proof of decidability in PSPACE, but also showed that the satisfiability of word equations can be decided in a non-deterministic linear space. At present, NP-hard [41] is the only known lower bound, which means that the question of whether the solution of word equations is NP-complete remains open.

In the context of word equations, we must mention quadratic word equations [84]. This specific fragment of word equations is characterized by the fact that each variable can occur at most twice. Most SMT solvers utilize a decision procedure based on Levi's lemma [64] to solve these equations, which has PSPACE complexity. However, it has been proven in [84] that solving quadratic word equations is NP-hard in general, even in the case where only one equation is involved, which was proved by a reduction from 3-SAT.

There are several significant extensions to the standard word equations that restrict the set of words that can be assigned to string variables. These important restrictions include memberships, relational constraints, arithmetic constraints over string lengths, and string-to-number conversion. In 1990, Schulz [88] showed that decidability of the problem is preserved if the string variables are additionally constrained by a regular language. However, the resulting complexity of such a problem is PSPACE-complete [24]. As for extensions with arithmetic constraints over string variables of the form $|x| = |y|$, it is still a long-standing open problem whether word equations with such constraints are decidable or not. Nevertheless, it is known that counting letters (e.g., counting the number of occurrences of 0 and 1 separately) leads to undecidability [24]. If the conversion between number and string is added to these arithmetic constraints, the satisfiability of such an extension of word equations is undecidable [43, 44]. Although relational constraints play a key role in expressing many functions in string-manipulating programs (e.g., escaping functions, replace-all, etc.), their satisfiability in the context of string theory, which involves only relational constraints, is undecidable [69]. Even checking a simple formula of the form $\mathcal{T}(x, x)$, for a given rational transducer $\mathcal{T}$, can easily be encoded into Post's correspondence problem [72]. Decidability is not obtained even if we use synchronized relational constraints, since the satisfiability of string constraints of the form $x = y \circ z \wedge \mathcal{T}(x, z)$; where $\mathcal{T}$ is a synchronized transducer, is undecidable [12]. Thus, extending word equations with relational constraints, without any restriction of such constraints, also leads to undecidability.

In the following sections we describe in detail two important fragments whose decision procedures are decidable, namely the straight-line fragment and the acyclic form.

3.2.1 Acyclic Form

In the paper [6] from 2014, a fragment of string logic named *Acyclic Form* was introduced. This fragment contains three types of constraints: word equations, arithmetic constraints over string variables, and membership constraints. Even though the decidability problem of such logic remains open [24], the Acyclic Form becomes decidable due to the specific constraints applied to the occurrence of string variables in word equations. The authors argue that this fragment is robust enough to cover all practical examples known at the time of this paper. Using the Acyclic Form, it was possible to verify the properties of implementations of common string-manipulating functions such as Hamming and Levenshtein distance.

The paper [6] also presents a decision procedure, which is both correct and complete; however, this is only valid for formulas that are in Acyclic Form. The foundation of this procedure lies in a set of inference rules. These rules replace literals of the input formula

with a set of new literals that simplify the original literal. It has been proven that the Acyclic Form is preserved after the application of any inference rule. The paper [6] further defines four groups of inference rules that must be applied in a precisely determined order to ensure the termination of the procedure. It is worth noting that paper [6] also introduces an innovative technique referred to as *splitting automata*. This method allows to deal with membership constraints of type $\mathcal{A}(t_{str} \circ t'_{str})$. Furthermore, this approach was used in other string solvers such as OSTRICH [32, 31, 30], Z3STR3RE [19, 17], SLOTH [49] and to some extent TRAU [4, 5, 8, 1, 2]. It should be noted that this method was also adapted for transducers [69]. The decision procedure was integrated into the string solver named NORN [6, 7], which is based on the DPLL(T) architecture.

Dependency graph [6]. Given a conjunction ϕ involving m (dis)equalities, we can build a *dependency graph* $G_\phi = (N, E, \texttt{label}, \texttt{map})$ in the following way. We order the (dis)equalities from e_1 to e_m, where each e_j is of the form $\texttt{lhs}(j) \approx \texttt{rhs}(j)$ for $j\colon 1 \leq j \leq m$ and $[\approx] \in \{=, \neq\}$. For each $j\colon 1 \leq j \leq m$, a node n_{2j-1} is used to refer to the left-hand side of the jth (dis)equality, and n_{2j} to its right-hand side. For example, two different nodes are used even in the case of the simple equality $u = u$, one to refer to the left-hand side, and the other to refer the right-hand side. N is then the set of $2 \times m$ nodes $\{n_i \mid i\colon 1 \leq i \leq 2 \times m\}$. The mapping $\texttt{label}$ associates the term $\texttt{lhs}(j)$ (resp. $\texttt{rhs}(j)$) to each node n_{2j-1} (resp. n_{2j}) for $j\colon 1 \leq j \leq m$. $\texttt{label}$ is not necessarily a one to one mapping. The mapping $\texttt{map}\colon E \to \{rel, var\}$ labels edges as follows: $\texttt{map}(n, n') = \texttt{rel}$ for each $(n, n') = (n_{2j-1}, n_{2j})$ for each $j\colon 1 \leq j \leq m$, and $\texttt{map}(n, n') = \texttt{var}$ iff $n \neq n'$, and $\texttt{label}(n)$ and $\texttt{label}(n')$ have some common variables. By construction, map is defined to be total, i.e., E contains only edges that are labeled by $\texttt{map}$.

Dependency cycle. Given a graph $G_\phi = (N, E, \texttt{label}, \texttt{map})$, a dependency cycle is defined as a sequence of distinct nodes $n_0, n_1, \ldots, n_k$ in N where $k \geq 1$. For this sequence, it must hold that $\forall i\colon 0 \leq i \leq k$, the mapping $\texttt{map}(n_i, n_{i+1\%(k+1)})$ must be defined and $\forall i\colon 0 \leq i < k, map(n_i, n_{i+1}) \neq \texttt{map}(n_{i+1}, n_{i+2\%(k+1)})$.

Acyclic form. A formula ϕ is said to be in *acyclic form* if and only if, no variable appears more than once in any equality or disequality in ϕ, and its dependency graph does not contain any dependency cycle.

3.2.2 Straight-Line Fragment

The straight-line fragment represents a decidable subclass of string logic, as initially introduced in [69]. This fragment includes word equations, relational constraints and membership constraints. Due to the properties of this fragment, it can be effectively utilized in modelling the logic of programs working with strings, especially those consisting of a sequence of simple steps executed sequentially (so-called straight-line programs). Such programs are often found during bounded model checking or dynamic symbolic execution, which unrolls loops in programs (up to a certain depth) and converts the resulting programs into SSA form, where each variable is defined only once. In [69], it is shown that the decidability of this fragment is preserved even when it is further extended to include arithmetic constraints over string variables, inequalities, letter counting, and IndexOf constraint. The authors further argue that the class known as *solved forms* [44] is essentially a subset of this extended fragment.

Dependency graph. Given a relational constraint ϕ, the *dependency graph* $G(\phi)$ of ϕ is the *directed* graph whose nodes are the string variables appearing in ϕ and there is an edge

from variable x to y iff (a) ϕ contains a conjunct of the form $R(x, y)$ for a rational relation R, or (b) an equation of the form $y = x_1 \ldots x_n$ for some string variables $x_1 \ldots x_n$ that include x.

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 $x_i, \ldots, x_m$.

Example 3.2.1. The program snippet in Example 1.0.1 would be expressed as $x = \mathcal{R}_1(\mathsf{name}) \wedge y = \mathcal{R}_2(x) \wedge z = w_1 \circ y \circ w_2 \circ x \circ w_3 \wedge u = \mathcal{R}_3(z)$. 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 an attack pattern `e1` is added as a conjunct. □

In the paper [69], a decision procedure for this fragment was proposed, where it was proven to be both sound and complete. It was also determined that the upper bound complexity of this procedure is EXPSPACE-complete. However, the authors mention that this complexity can be reduced to PSPACE under a certain reasonable assumption (see theorem 10 in [69]). Although the paper [69] provides a theoretical proof of decidability and an upper bound on the complexity, it unfortunately does not offer a concrete implementable solution to this procedure. This deficiency is addressed with the advent of a string solver called Sloth [49], which reduces the satisfiability of formulas in a straight-line fragment to the emptiness problem of alternating finite-state automata (AFAs). It should be noted that the reduction is at worst exponential with respect to the number of concatenation operations, which is consistent with the previously identified computational limitation imposed by the EXPSPACE-hardness of the problem [69]. Yet, it is important to emphasize that, except in extreme cases, the reduction is polynomial with respect to the size of the given formula. Subsequently, a new efficient decision procedure for this fragment was introduced and implemented in the new fast string solver OSTRICH [32, 31, 30].

In the context of straight-line fragment, the paper [29] focused on the decidability boundaries of this fragment that includes the `replaceAll` function along with regular constraints. The authors proved that if string variables are used as pattern parameters in the `replaceAll` function, then this string theory becomes undecidable (a reduction from Post's correspondence problem). However, if the pattern parameters of the `replaceAll` function are regular expressions, the satisfiability of the given chain theory is decidable with complexity determined as EXPSPACE. Moreover, the authors showed that the satisfiability problem is PSPACE-complete for several practical scenarios (Corollary 4.7 in [29]). If we extend the string theory with a constant letter as the pattern parameter for the `replaceAll` function and include arithmetic constraints over string variables, the satisfiability again becomes undecidable. Undecidability can also be achieved using integer constraints, character constraints, or constraints involving the `IndexOf` function.

Chapter 4

Contributions

In this chapter, we provide a brief summary of the individual papers that form the main part of this thesis. The individual articles mentioned here can be found in Appendix A.

4.1 String Constraints with Concatenation and Transducers Solved Efficiently

In this section, we summarize our work [49], which is attached in Appendix A. The main technical contribution 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 [69, 6] 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 $\mathcal{A}_1$ and $\mathcal{A}_2$ would be of size $O(|\mathcal{A}_1| \times |\mathcal{A}_2|)$, the language can be represented using an AFA of size $|\mathcal{A}_1| + |\mathcal{A}_2|$ (e.g. see [99]). 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 $\mathcal{A}$) was reduced in [69, 6] to regular constraints on y and z by means of splitting $\mathcal{A}$, which causes a cubic blow-up (since an "intermediate state" in $\mathcal{A}$ 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 $\mathcal{T}$ gives us an automaton of size $O(|\mathcal{T}| \times |\mathcal{A}|)$. 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 [15, 86, 12]). 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 similar to the concept of alternating symbolic automata in [37] 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 [61], 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 [103, 36], which can be solved by state of the art model checking algorithms, such as IC3 [22], k-induction [93], or Craig interpolation-based methods [71], and tools like nuXmv [28] or ABC [23].

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 [12, 13], 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 [85], and applying the nuXmv [28] and ABC [23] 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`[1], 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 1.0.1. Experiments with string benchmarks drawn from the literature, including problems with `replace`, `replaceAll`, 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.

4.2 Chain-Free String Constraints

In this section, we provide an overview of our work [8] attached in Appendix A. In the presented text, we propose an approach that combines two research directions: finding decidable fragments and utilizing them to develop efficient semi-algorithms. To that aim, we define the class of *chain-free* formulas which strictly subsumes the acyclic fragment of Norn [7] as well as the straight-line fragment of [69, 49, 29], and thus extends the known border of decidability for string constraints. The extension is of a practical relevance. A straight-line constraint models a path through a string program in the single static assignment form, but as soon as the program compares two initialised string variables, the string constraint falls out of the fragment. The acyclic restriction of Norn on the other hand does not include transducer constraints (although it might be extended to them) and does not allow multiple occurrences of a variable in a single string constraint (e.g. an equation

[1] `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.

of the form $x \circ y = z \circ z$). Our chain-free fragment is liberal enough to accommodate constraints that share both these forbidden features (including $x \circ y = z \circ z$).

The main idea behind the chain-free fragment is to associate to the set of relational constraints a *splitting graph* where each node corresponds to an occurrence of a variable in the relational constraints of the formula (as shown in Figure 4.1). An edge from an occurrence of x to an occurrence of y means that the source occurrence of x appears in a relational constraint which has in the opposite side an occurrence of y different from the target occurrence of y. The chain-free fragment prohibits loops in the graph, that we call *chains*, such as those shown in red in Figure 4.1.

Then, we identify the so called *weakly chaining* fragment which strictly extends the chain-free fragment by allowing *benign* chains. Benign chains relate relational constraints where each left side contains only one variable, the constraints are all *length preserving*, and all the nodes of the cycles appear exclusively on the left or exclusively on the right sides of the involved relational constraints (as is the case in Figure 4.1). Weakly chaining constraints may in practice arise from the checking that an encoding followed a decoding function is indeed the identity, i.e., satisfiability of constraints of the form $\mathcal{T}_{\text{enc}}(\mathcal{T}_{\text{dec}}(x)) = x$, discussed e.g. in [52]. For instance, in situations similar to the example 4.2.1, one might like to verify that the sanitization of a password followed by the application of a function supposed to invert the sanitization gives the original password. The weakly chaining fragment is then formally defined as follows:

Splitting graph. Let $\Psi ::= \bigwedge_{j=1}^{m} \varphi_j$ be a conjunction of relational string constraints with $\varphi_j ::= R_j(t_{2j-1}, t_{2j}), 1 \leq j \leq m$ where for each $i : 1 \leq i \leq 2m$, t_i is a concatenation of variables $x_i^1 \circ \cdots \circ x_i^{n_i}$. We define the set of *positions* of Ψ as $P = \{(i,j) \mid 1 \leq j \leq 2m \wedge 1 \leq i \leq n_j\}$. The *splitting graph* of Ψ is then the graph $G_\Psi = (P, E, \mathtt{var}, \mathtt{con})$ where the positions in P are its nodes, and the mapping $\mathtt{var} : P \to \mathbb{X}$ labels each position (i,j) with the variable x_j^i appearing at that position. We say that $(i, 2j-1)$ (resp. $(i, 2j)$) is the ith *left* (resp. *right*) positions of the jth constraint, and that R_j is the predicate of these positions. Any pair of a left and a right position of the same constraint are called *opposing*. The set of edges E then consists of edges (p, p') between positions for which there is an intermediate position p'' (different from p') that is opposing to p and is labeled by the same variable as p' ($\mathtt{var}(p'') = \mathtt{var}(p')$). Finally, the labelling $\mathtt{con}$ of edges assigns to (p, p') the constraint of p, that is, $\mathtt{con}(p, p') = R_j$ where p is a position of the jth constraint. An example of a splitting graph is on Fig. 4.1.

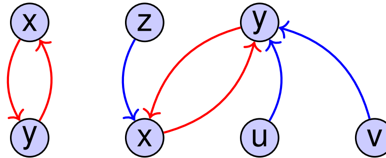


Figure 4.1: The splitting graph of $x = z \cdot y \wedge y = x \cdot u \cdot v$.

Chains. A *chain*[2] is a sequence of the form $(p_0, p_1), (p_1, p_2), \ldots, (p_n, p_0)$ of edges in E. A chain is *benign* if (1) all the relational constraints corresponding to the edges $\mathtt{con}(p_0, p_1)$, $\mathtt{con}(p_1, p_2), \ldots, \mathtt{con}(p_n, p_0)$ are left sided and and all the string relations involved in these constraints are length preserving, and (2) the sequence of positions $p_0, p_1, \ldots, p_n$ consists of

[2] We use chains instead of cycles in order to avoid confusion between our decidable fragment and the ones that exist in literatures.

left positions only, or from right positions only. Observe that if there is a benign chain that uses only right positions then there exists also a benign chain that uses only left positions. The graph is *chain-free* if it has no chains, and it is *weakly chaining* if all its chains are benign. A formula is *chain-free* (resp. *weakly chaining*) if the splitting graph of every clause in its DNF is chain-free (resp. weakly chaining). Benign chains are on Fig. 4.1 shown in red.

Example 4.2.1. The following pseudo-PHP code (a variation of a code at [98]) that prompts a user to change his password is an example of a program that generates a chain-free constraint that is neither straight-line nor acyclic form according to [69, 6].

```
$old=$database->real_escape_string($oldIn);
$new=$database->real_escape_string($newIn);
$pass=$database->query("SELECT password FROM users WHERE userID=".$user);
if($pass == $old)
   if($new != $old)
      $query = "UPDATE users SET password=".$new." WHERE userID=".$user;
      $database->query($query);
```

The user inputs the old password `oldIn` and the new password `newIn`, both are sanitized and assigned to `old` and `new`, respectively. The old sanitized password is compared with the value `pass` from the database, to authenticate the user, and then also with the new sanitized password, to ensure that a different password was chosen, and finally saved in the database. The sanitization is present to prevent SQL injection. To ensure that the sanitization works, we wish to verify that the SQL query `query` is safe, that is, it does not belong to a regular language *Bad* of dangerous inputs. This safety condition is expressed by the constraint

$$\texttt{new} = \mathcal{T}(\texttt{newIn}) \land \texttt{old} = \mathcal{T}(\texttt{oldIn}) \land \texttt{pass} = \texttt{old} \land \texttt{new} \neq \texttt{old} \\ \land \texttt{query} = u.\texttt{new}.v.\texttt{user} \land \texttt{query} \in \mathit{Bad}$$

The sanitization on lines 1 and 2 is modeled by the transducer $\mathcal{T}$, and u and v are the constant strings from line 7. The constraints fall out from the straight-line due to the test $\texttt{new} \neq \texttt{old}$. □

Our decision procedure for the weakly chaining formulas proceeds in several steps. The formula is transformed to an equisatisfiable chain-free formula, and then to an equisatisfiable concatenation free formula in which the relational constraints are of the form $\mathcal{T}(x, y)$ where x and y are two string variables and $\mathcal{T}$ is a transducer/relational constraint. Finally, we provide a decision procedure of a chain and concatenation-free formulae. The algorithm is based on two techniques. First, we show that the chain-free conjunction over relational constraints can be turned into a single equivalent transducer constraint (in a similar manner as in [12]). Second, consistency of the resulting transducer constraint with the input length constraints is checked via the computation of the Parikh image of the transducer.

To demonstrate the usefulness of our approach, we have implemented our decision in Sloth [49], and then integrated it in the open-source solver Trau [4, 5]. Trau is a string solver which is based on a Counter-Example Guided Abstraction Refinement (CEGAR) framework which contains both an under- and an over-approximation module. These two modules interact together in order to automatically make these approximations more precise. We have implemented our decision procedure inside the over-approximation module which takes as an input a constraint and checks if it belongs to the weakly chaining fragment. If it is the case, then we use our decision procedure outlined above. Otherwise, we start by choosing a minimal set of occurrences of variables x that needs to be replaced by

fresh ones such that the resulting constraint falls in our decidable fragment. We compare our prototype implementation against four other state-of-the-art string solvers, namely OSTRICH [32], Z3str3 [18], cvc4 [66], and Trau [3]. For our comparison with Z3str3, we use the version that is part of Z3 4.8.4. Our experimental results show the competitiveness as as well as accuracy of the framework compared to the solver Trau [4, 5]. Furthermore, the experimental results show the competitiveness and generality of our method compared to the existing techniques.

4.3 Efficient Handling of String-Number Conversion

This section summarizes the paper presented in [1], which is attached in Appendix A. In the given paper, we propose a framework that efficiently handles string constraints with string-number conversion. Since the problem is provably unsolvable, our framework combines over and under-approximation techniques. The over-approximation is for proving UNSAT when possible, while the under-approximation is for proving SAT when possible. Both over- and under-approximation fall in a decidable fragment of string constraints that we can efficiently solve.

For ease of presentation, we use the following toy example

$$\Phi = \{\text{"0"}x = x\text{"0"}, \mathsf{toNum}(x) = \mathsf{toNum}(y), |y| > |x| > 1, 1000 < |y|\}$$

to explain the main ideas behind our decision procedure. To make our terminology explicit: Φ states that "0" concatenated with x is the same as x with "0", the numeric value of the string x is equivalent to that of y, y is longer than x, y is longer than 1000 characters, and x is longer than 1. Notice that Φ is satisfiable. E.g., it has a model x = "00" and y = "0^{1002}". Although this toy example is seemingly trivial, all the state-of-the-art string constraint solvers we tried (including Z3, cvc4, and Z3str3) cannot solve it within 10 minutes.

Our new decision procedure solves the example in few seconds. It proceeds in two steps: the first step consists in over-approximating the set of input constraints into a set that falls in the chain-free fragment [8], which is decidable. Observe that we could over-approximate the input constraint into any decidable fragment, e.g. the acyclic form [6] or the straight-line fragment [32]. Our choice of the chain-free fragment [8] is only motivated by the fact that the chain-free fragment is the *largest* known decidable fragment for that class of string constraints. In our example, we over-approximate the formula Φ by converting "0"$x = x$"0" to two formulae $\{x_1 = \text{"0"}x, x_2 = x\text{"0"}\}$ and replacing the constraint $\mathsf{toNum}(x) = \mathsf{toNum}(y)$ with $n_x = n_y \wedge (n_x = -1 \vee (n_x \neq -1 \wedge x \in [0-9]^*)) \wedge (n_y = -1 \vee (n_y \neq -1 \wedge y \in [0-9]^*)))$. Observe that if the over-approximation is UNSAT then our decision procedure declares that the original formula is also UNSAT and terminates. Surprisingly, despite its simplicity, our over-approximation procedure works very well in practice as shown by our experimental results. Coming back to the formula Φ, the over-approximation module will return SAT in this case.

The second step of our decision procedure is only enabled if the over-approximation step returns SAT. In this case, our decision procedure uses an under-approximation technique (which is our main contribution) to restrict the search domain of each string variable to strings that obey some predefined and parameterized pattern. We propose to use patterns defined by *parametric flat automata* (PFA). A PFA is a *flat* finite state automaton consisting of a predefined sequence of loops, each of fixed length (see Figure 4.2). The size of the PFA is parameterized by the length of the sequence of loops and the size of each

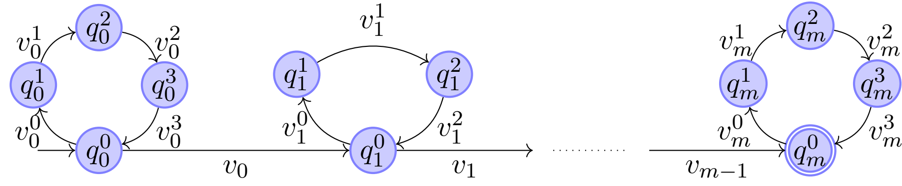


Figure 4.2: An example of a parametric flat automaton

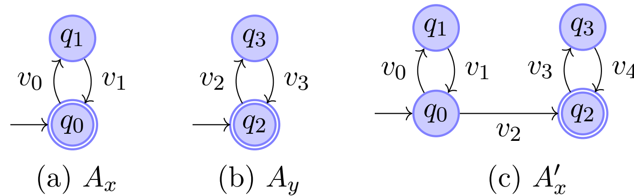
loop. Adjusting these parameters enlarges or prunes the potential solution space. This approach based on PFA is very flexible yet allows very efficient manipulation. In fact, our procedure restricts the search space for each variable to the set of words accepted by the corresponding given PFA.

Then, we show that given such restriction, one can reduce the string constraint solving problem to a linear formula satisfiability problem in polynomial-time. To gain in efficiency, we label each transition of a PFA with a unique *character* variable (whose domain is the set of natural numbers) instead of having a transition between every two states for each symbol in the alphabet. This is done by associating to each character in our alphabet a unique natural number. This allows us to avoid the *alphabet explosion problem* from which the approach in [4] suffers and it is the key for handling string-number conversion efficiently.

In the following, we explain the construction of the linear formula using Φ as an example. Assume that we project the domains of x and y to the PFA in Figure 4.3 (a) and (b), respectively. The variables v_0, v_1, v_2, v_3 in the figure are *character* variables. Thus, v_0, v_1, v_2, v_3 are also integer variables.

The linear formula produced after the domain restriction will be over variables v_0, v_1, v_2, v_3, as well as the number of occurrences of each character variable $\#v_0$, $\#v_1$, $\#v_2$, $\#v_3$. Each model of the linear formula encodes a model of the string constraint. For example, $x =$ "00" and $y =$ "0^{1002}" is encoded by the assignment $(v_0, v_1, v_2, v_3, \#v_0, \#v_1, \#v_2, \#v_3) \to (0, 0, 0, 0, 1, 1, 501, 501)$.[3] The assignment says, for example, that x is the *parametric word* obtained by traversing the loop of A_x once (because $\#v_0 = \#v_1 = 1$), which is $v_0 v_1$. Under the assignment $v_0 = 0$ and $v_1 = 0$, we obtain $x =$ "00".

If a model of the produced linear formula is found, then the procedure concludes SAT with an assignment to the string variables. If not, our procedure changes the PFAs to a more expressive one (by adding more states and transitions) and repeat the analysis. We report unknown after failing to prove SAT using a certain number of PFAs.



(a) A_x (b) A_y (c) A'_x

Figure 4.3: Parametric flat automata of x and y

To demonstrate the usefulness of our approach, we have implemented our decision procedure in an open source solver, called Z3-Trau and evaluated it on a large set of benchmarks

[3] In these examples, we use the shorthand $(x_1, \ldots, x_k) \to (n_1, \ldots, n_k)$ to denote the function $\{x_1 \mapsto n_1, \ldots, x_k \mapsto n_k\}$.

obtained from the literature and from symbolic execution of real world programs. The experimental results show that Z3-Trau is among the best tools for solving basic string constraints and significantly outperforms all other tools on benchmarks with string-number conversion constraints. In this benchmark, the total amount of tests cannot be solved by Z3-Trau is only a half to the second best tool.

4.4 Solving String Constraints with Approximate Parikh Image

In the following text, we provide a summary of our paper [55], which can be found in Appendix A. In the mentioned paper, we introduce an extension to the decision procedure for the straight-line fragment. This extension is integrated into the string solver Sloth [49], as the original procedure, described in Section 4.1, struggles with solving arithmetic constraints over strings. Unfortunately, the mentioned procedure does not allow the use of the standard Parikh image-based method for handling arithmetic constraints over strings, which is typically used in nondeterministic finite automata.

Our extension for solving arithmetic constraints over strings works as follows. First, for each AFA, we construct its Parikh image. To do this, we use a modified version of the algorithm described in [101], originally designed for calculating the Parikh image of a context-free grammar. We then perform the product of these obtained Parikh images. Since Sloth uses AFAs with symbolically represented transitions (transitions can have a set of symbols on the edge instead of one symbol), we proposed a new method for dealing with Parikh images that allows us to perform the product of two different Parikh images. Since the Parikh image does not support preserving the order of symbols in a word, our solution is thus an over-approximation. Subsequently, we integrated this extension into Sloth. The experimental results showed that our extension of Sloth has good results on both simple benchmarks and complex benchmarks that are real-word combinations of transducers and concatenation constraints.

Chapter 5

Conclusions and Future Directions

In this conclusion, we summarize the main points of this thesis and briefly outline possible directions for future research.

5.1 Summary of the Contributions

In this thesis, we reviewed our results in the field of string constraints solving. First, we have discussed in detail the practical motivations driving the research in the field of security, especially in the context of string manipulation. Such manipulations can be the source of serious security threats, such as Cross-Site Scripting (XSS) and SQL injection. Subsequently, we introduced existing decision procedures and their complexities, where we discussed in detail two important decidable fragments, namely the acyclic form and the straght-line fragment. We then summarized the main contribution of this work: first, we proposed a fast reduction of the satisfiability of formulae in the straight-line and acyclic fragments to the emptiness problem of alternating finite-state automata (AFA), which is polynomial in most cases. This reduction, in combination with advanced model checking algorithms such as IC3, provides the first practical algorithm for solving string constraints involving concatenation, finite-state transducers and regular constraints. Second, we introduced a new fragment of string constraints called chain-free and its relaxation called weakly chaninng, along with decision procedures for these fragments. It is important to mention that these new fragments generalize both the straight-line fragment and the acyclic form. Third, we presented a method for checking the satisfiability of string constraints, in particular with string-to-number conversion, using parametric flat automata (PFA). This procedure was complemented by an algorithm for converting string constraints to linear formulas in polynomial time with a search space bounded by PFA. Lastly, we proposed and integrated an improved Parikh abstraction into the string solver SLOTH for solving length constraints.

5.2 Further Directions

The results of this thesis have been successfully developed and utilized, especially within the work [21]. In the mentioned work, the string solver called Noodler was developed, which is based on the theory behind the chain-free fragment. This solver is one of the most efficient string solvers, capable of outperforming industrial solvers.

In the future, we plan to focus our upcoming research on expanding support for other practically relevant operations, such as splitting at delimiters and `indexOf`. Furthermore, we would like to focus on improving the performance of SLOTH through improved algorithms for alternating automata and and through optimising the automata encodings of string problems. Additionally, we intend to extend our approach to a more general class of length constraints (e.g. Presburgerexpressible constraints). However, this extension seems to be rather challenging since it would require extending alternating finite automata with monotonic counters (see [69]), for which efficiently solving language emptiness is a difficult open problem.

Regarding the further development of the Z3-TRAU, we would like to integrate it with a symbolic executor for JavaScript. From a technical perspective, we believe that it would be interesting to consider (symbolic) flattening of an even larger set of string operations, such as the one containing `replaceAll` and `split`.

Bibliography

[1] Abdulla, P. A., Atig, M. F., Chen, Y., Diep, B. P., Dolby, J., Janku, P., Lin, H., Holík, L. and Wu, W. Efficient handling of string-number conversion. In: Donaldson, A. F. and Torlak, E., ed. *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020.* Association for Computing Machinery, 2020, p. 943–957. DOI: 10.1145/3385412.3386034. ISBN 9781450376136. Available at: https://doi.org/10.1145/3385412.3386034.

[2] Abdulla, P. A., Atig, M. F., Chen, Y.-F., Diep, B. P., Holík, L., Hu, D., Tsai, W.-L., Wu, Z. and Yen, D.-D. Solving Not-Substring Constraint with Flat Abstraction. In: Oh, H., ed. *Programming Languages and Systems: 19th Asian Symposium, APLAS 2021, Chicago, IL, USA, October 17–18, 2021, Proceedings 19.* 2021, p. 305–320. ISBN 978-3-030-89051-3.

[3] Abdulla, P. A., Atig, M. F., Chen, Y., Diep, B. P., Holík, L., Rezine, A. and Rümmer, P. *Trau String Solver* [https://github.com/diepbp/FAT]. Available at: https://github.com/diepbp/FAT.

[4] Abdulla, P. A., Atig, M. F., Chen, Y., Diep, B. P., Holík, L., Rezine, A. and Rümmer, P. Flatten and conquer: a framework for efficient analysis of string constraints. In: Cohen, A. and Vechev, M. T., ed. *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017.* ACM, 2017, p. 602–617. ISBN 9781450349888.

[5] Abdulla, P. A., Atig, M. F., Chen, Y.-F., Diep, B. P., Holík, L., Rezine, A. and Rümmer, P. Trau: SMT solver for string constraints. In: IEEE. *2018 Formal Methods in Computer Aided Design (FMCAD).* 2018, p. 1–5.

[6] Abdulla, P. A., Atig, M. F., Chen, Y.-F., Holík, L., Rezine, A., Rümmer, P. and Stenman, J. String constraints for verification. In: Biere, A. and Bloem, R., ed. *Computer Aided Verification: 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings 26.* 2014, p. 150–166. ISBN 978-3-319-08867-9.

[7] Abdulla, P. A., Atig, M. F., Chen, Y.-F., Holík, L., Rezine, A., Rümmer, P. and Stenman, J. Norn: An SMT solver for string constraints. In: Springer. *International conference on computer aided verification.* 2015, p. 462–469. ISBN 978-3-319-21690-4.

[8] Abdulla, P. A., Atig, M. F., Diep, B. P., Holík, L. and Janků, P. Chain-free string constraints. In: Chen, Y.-F., Cheng, C.-H. and Esparza, J., ed. *Automated Technology for Verification and Analysis: 17th International Symposium, ATVA 2019, Taipei, Taiwan, October 28–31, 2019, Proceedings 17*. 2019, p. 277–293. ISBN 978-3-030-31784-3.

[9] Amadini, R., Gange, G., Stuckey, P. J. and Tack, G. A Novel Approach to String Constraint Solving. In: Beck, J. C., ed. *Principles and Practice of Constraint Programming*. Cham: Springer International Publishing, 2017, p. 3–20. ISBN 978-3-319-66158-2.

[10] Aydin, A., Bang, L. and Bultan, T. Automata-based model counting for string constraints. In: Springer. *International Conference on Computer Aided Verification*. 2015, p. 255–272.

[11] Barbosa, H., Barrett, C., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A. et al. Cvc5: A versatile and industrial-strength SMT solver. In: Springer. *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 2022, p. 415–442.

[12] Barceló, P., Figueira, D. and Libkin, L. Graph Logics with Rational Relations. *Logical Methods in Computer Science*. 2013, vol. 9, no. 3. DOI: 10.2168/LMCS-9(3:1)2013.

[13] Barceló, P., Libkin, L., Lin, A. W. and Wood, P. T. Expressive Languages for Path Queries over Graph-Structured Data. *ACM Trans. Database Syst.* 2012, vol. 37, no. 4, p. 31.

[14] Barrett, C., Tinelli, C., Deters, M., Liang, T., Reynolds, A. and Tsiskaridze, N. Efficient solving of string constraints for security analysis. In: *Proceedings of the Symposium and Bootcamp on the Science of Security*. 2016, p. 4–6.

[15] Berstel, J. *Transductions and Context-Free Languages*. Teubner-Verlag, 1979.

[16] Berzish, M. *Z3str4: A Solver for Theories over Strings*. 2021. Dissertation. University of Waterloo, Ontario, Canada. Available at: https://hdl.handle.net/10012/17102.

[17] Berzish, M., Day, J. D., Ganesh, V., Kulczynski, M., Manea, F., Mora, F. and Nowotka, D. Towards more efficient methods for solving regular-expression heavy string constraints. *Theoretical Computer Science*. Elsevier. 2023, vol. 943, p. 50–72.

[18] Berzish, M., Ganesh, V. and Zheng, Y. Z3str3: A string solver with theory-aware heuristics. In: IEEE. *2017 Formal Methods in Computer Aided Design (FMCAD)*. 2017, p. 55–59.

[19] Berzish, M., Kulczynski, M., Mora, F., Manea, F., Day, J. D., Nowotka, D. and Ganesh, V. An SMT solver for regular expressions and linear arithmetic over string length. In: Springer. *International Conference on Computer Aided Verification*. 2021, p. 289–312.

[20] Bjørner, N., Tillmann, N. and Voronkov, A. Path feasibility analysis for string-manipulating programs. In: Springer. *International Conference on Tools and Algorithms for the Construction and Analysis of Systems.* 2009, p. 307–321.

[21] Blahoudek, F., Chen, Y.-F., Chocholatỳ, D., Havlena, V., Holík, L., Lengál, O. and Síč, J. Word Equations in Synergy with Regular Constraints. In: Springer. *International Symposium on Formal Methods.* 2023, p. 403–423.

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

[23] Brayton, R. and Mishchenko, A. ABC: An Academic Industrial-Strength Verification Tool. In: Touili, T., Cook, B. and Jackson, P., ed. *Computer Aided Verification: 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, p. 24–40. DOI: 10.1007/978-3-642-14295-6_5. ISBN 978-3-642-14295-6. Available at: http://dx.doi.org/10.1007/978-3-642-14295-6_5.

[24] Büchi, J. R. and Senger, S. Definability in the existential theory of concatenation and undecidable extensions of this theory. In: *The Collected Works of J. Richard Büchi.* Springer, 1990, p. 671–683.

[25] Cadar, C., Dunbar, D., Engler, D. R. et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: *OSDI.* 2008, vol. 8, p. 209–224.

[26] Cadar, C., Ganesh, V., Pawlowski, P. M., Dill, D. L. and Engler, D. R. EXE: Automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC).* ACM New York, NY, USA. 2008, vol. 12, no. 2, p. 1–38.

[27] Cadar, C. and Sen, K. Symbolic execution for software testing: three decades later. *Communications of the ACM.* ACM New York, NY, USA. 2013, vol. 56, no. 2, p. 82–90.

[28] Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M. and Tonetta, S. The nuXmv Symbolic Model Checker. In: *CAV'14.* Springer, 2014, vol. 8559, p. 334–342. Lecture Notes in Computer Science.

[29] Chen, T., Chen, Y., Hague, M., Lin, A. W. and Wu, Z. What is Decidable About String Constraints with the ReplaceAll Function. *Proc. ACM Program. Lang.* New York, NY, USA: ACM. december 2018, vol. 2, POPL. DOI: 10.1145/3158091. ISSN 2475-1421. Available at: http://doi.acm.org/10.1145/3158091.

[30] Chen, T., Flores Lamas, A., Hague, M., Han, Z., Hu, D., Kan, S., Lin, A. W., Rümmer, P. and Wu, Z. Solving string constraints with regex-dependent functions through transducers with priorities and variables. *Proceedings of the ACM on Programming Languages.* ACM New York, NY, USA. 2022, vol. 6, POPL, p. 1–31.

[31] CHEN, T., HAGUE, M., HE, J., HU, D., LIN, A. W., RÜMMER, P. and WU, Z. A decision procedure for path feasibility of string manipulating programs with integer data type. In: Springer. *International Symposium on Automated Technology for Verification and Analysis*. 2020, p. 325–342. ISBN 978-3-030-59152-6.

[32] CHEN, T., HAGUE, M., LIN, A. W., RÜMMER, P. and WU, Z. Decision Procedures for Path Feasibility of String-manipulating Programs with Complex Operations. *Proc. ACM Program. Lang.* New York, NY, USA: ACM. january 2019, vol. 3, POPL. DOI: 10.1145/3290362. ISSN 2475-1421. Available at: http://doi.acm.org/10.1145/3290362.

[33] CHEN, Y., HAVLENA, V., LENGÁL, O. and TURRINI, A. A Symbolic Algorithm for the Case-Split Rule in String Constraint Solving. In: S. OLIVEIRA, B. C. d., ed. *Programming Languages and Systems - 18th Asian Symposium, APLAS 2020, Fukuoka, Japan, November 30 - December 2, 2020, Proceedings*. Springer, 2020, vol. 12470, p. 343–363. Lecture Notes in Computer Science. DOI: 10.1007/978-3-030-64437-6_18. Available at: https://doi.org/10.1007/978-3-030-64437-6_18.

[34] CHEN, Y.-F., HAVLENA, V., LENGÁL, O. and TURRINI, A. A symbolic algorithm for the case-split rule in solving word constraints with extensions. *Journal of Systems and Software*. 2023, vol. 201, p. 111673. DOI: https://doi.org/10.1016/j.jss.2023.111673. ISSN 0164-1212. Available at: https://www.sciencedirect.com/science/article/pii/S0164121223000687.

[35] CO, G. *Google Closure Library (referred in Aug 2023)*. *https://developers.google.com/closure/library/*. 2023.

[36] COX, A. and LEASURE, J. Model Checking Regular Language Constraints. *CoRR*. 2017, abs/1708.09073. Available at: http://arxiv.org/abs/1708.09073.

[37] D'ANTONI, L., KINCAID, Z. and WANG, F. A Symbolic Decision Procedure for Symbolic Alternating Finite Automata. *CoRR*. 2016, abs/1610.01722. Available at: http://arxiv.org/abs/1610.01722.

[38] DAY, J. D., EHLERS, T., KULCZYNSKI, M., MANEA, F., NOWOTKA, D. and POULSEN, D. B. On Solving Word Equations Using SAT. In: FILIOT, E., JUNGERS, R. M. and POTAPOV, I., ed. *Reachability Problems - 13th International Conference, RP 2019, Brussels, Belgium, September 11-13, 2019, Proceedings*. Springer, 2019, vol. 11674, p. 93–106. Lecture Notes in Computer Science. DOI: 10.1007/978-3-030-30806-3_8. Available at: https://doi.org/10.1007/978-3-030-30806-3_8.

[39] DE MOURA, L. and BJØRNER, N. Z3: An efficient SMT solver. In: Springer. *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. 2008, p. 337–340.

[40] DIEKERT, V. Makanin's Algorithm. In: LOTHAIRE, M., ed. *Algebraic Combinatorics on Words*. Cambridge University Press, 2002, vol. 90, chap. 12, p. 387–442. Encyclopedia of Mathematics and its Applications.

[41] Ehrenfreucht, A. and Rozenberg, G. Finding a homomorphism between two words in np-complete. *Information Processing Letters*. Elsevier. 1979, vol. 9, no. 2, p. 86–88.

[42] Fu, X. and Li, C. Modeling Regular Replacement for String Constraint Solving. In: *NFM'10*. 2010, NASA/CP-2010-216215, p. 67–76. NASA.

[43] Ganesh, V. and Berzish, M. Undecidability of a theory of strings, linear arithmetic over length, and string-number conversion. *ArXiv preprint arXiv:1605.09442*. 2016. Available at: http://arxiv.org/abs/1605.09442.

[44] Ganesh, V., Minnes, M., Solar Lezama, A. and Rinard, M. Word Equations with Length Constraints: What's Decidable? In: Biere, A., Nahir, A. and Vos, T., ed. *Hardware and Software: Verification and Testing*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, p. 209–226. ISBN 978-3-642-39611-3.

[45] Godefroid, P., Klarlund, N. and Sen, K. DART: Directed automated random testing. In: *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. 2005, p. 213–223.

[46] Gutiérrez, C. Satisfiability of word equations with constants is in exponential space. In: IEEE. *Proceedings 39th Annual Symposium on Foundations of Computer Science (Cat. No. 98CB36280)*. 1998, p. 112–119. DOI: 10.1109/SFCS.1998.743434.

[47] Gutiérrez, C. Solving Equations in Strings: On Makanin's Algorithm. In: *LATIN*. 1998, p. 358–373.

[48] Heiderich, M., Schwenk, J., Frosch, T., Magazinius, J. and Yang, E. Z. Mxss attacks: Attacking well-secured web-applications by using innerhtml mutations. In: *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. 2013, p. 777–788.

[49] Holík, L., Janků, P., Lin, A. W., Rümmer, P. and Vojnar, T. String constraints with concatenation and transducers solved efficiently. *Proceedings of the ACM on Programming Languages*. ACM New York, NY, USA. 2018, vol. 2, POPL, p. 1–32.

[50] Hooimeijer, P., Livshits, B., Molnar, D., Saxena, P. and Veanes, M. Fast and Precise Sanitizer Analysis with {BEK}. In: *20th USENIX Security Symposium (USENIX Security 11)*. 2011.

[51] Hooimeijer, P. and Weimer, W. StrSolve: Solving string constraints lazily. *Autom. Softw. Eng.* 2012, vol. 19, no. 4, p. 531–559.

[52] Hu, Q. and D'Antoni, L. Automatic Program Inversion Using Symbolic Transducers. *SIGPLAN Not.* New York, NY, USA: ACM. june 2017, vol. 52, no. 6. ISSN 0362-1340.

[53] invicti. *An XSS Vulnerability is Worth up to $10,000 According to Google* [https://www.invicti.com/blog/web-security/google-increase-reward-vulnerability-program-xss/]. 2013.

[54] Jaffar, J. Minimal and complete word unification. *Journal of the ACM (JACM)*. ACM New York, NY, USA. 1990, vol. 37, no. 1, p. 47–85. ISSN 0004-5411.

[55] Janků, P. and Turoňová, L. Solving string constraints with approximate parikh image. In: Springer. *International Conference on Computer Aided Systems Theory.* 2019, p. 491–498.

[56] Jez, A. Recompression: a simple and powerful technique for word equations. In: Portier, N. and Wilke, T., ed. *30th International Symposium on Theoretical Aspects of Computer Science (STACS 2013).* Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2013, vol. 20, p. 233–244. Leibniz International Proceedings in Informatics (LIPIcs). DOI: 10.4230/LIPIcs.STACS.2013.233. ISBN 978-3-939897-50-7. Available at: http://drops.dagstuhl.de/opus/volltexte/2013/3937.

[57] Jez, A. Word equations in nondeterministic linear space. In: Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. *44th International Colloquium on Automata, Languages, and Programming (ICALP 2017).* 2017.

[58] Kern, C. Securing the tangled web. *Communications of the ACM.* ACM New York, NY, USA. 2014, vol. 57, no. 9, p. 38–47.

[59] Kiezun, A., Ganesh, V., Artzi, S., Guo, P. J., Hooimeijer, P. and Ernst, M. D. HAMPI: A solver for word equations over strings, regular expressions, and context-free grammars. *ACM Transactions on Software Engineering and Methodology (TOSEM).* ACM New York, NY, USA. 2013, vol. 21, no. 4, p. 1–28.

[60] King, J. C. Symbolic execution and program testing. *Communications of the ACM.* ACM New York, NY, USA. 1976, vol. 19, no. 7, p. 385–394.

[61] Klarlund, N., Møller, A. and Schwartzbach, M. I. MONA Implementation Secrets. *International Journal of Foundations of Computer Science.* World Scientific. 2002, vol. 13, no. 4, p. 571–586.

[62] Kościelski, A. and Pacholski, L. Complexity of Makanin's algorithm. *Journal of the ACM (JACM).* ACM New York, NY, USA. 1996, vol. 43, no. 4, p. 670–684. ISSN 0004-5411.

[63] Le, Q. L. and He, M. A decision procedure for string logic with quadratic equations, regular expressions and length constraints. In: Springer. *Programming Languages and Systems: 16th Asian Symposium, APLAS 2018, Wellington, New Zealand, December 2–6, 2018, Proceedings 16.* 2018, p. 350–372.

[64] Levi, F. W. On semigroups. *Bull. Calcutta Math. Soc.* 1944, vol. 36, 141-146, p. 82.

[65] Li, G. and Ghosh, I. PASS: String Solving with Parameterized Array and Interval Automaton. In: Bertacco, V. and Legay, A., ed. *Hardware and Software: Verification and Testing.* Cham: Springer International Publishing, 2013, p. 15–31. ISBN 978-3-319-03077-7.

[66] Liang, T., Reynolds, A., Tinelli, C., Barrett, C. and Deters, M. A DPLL (T) theory solver for a theory of strings and regular expressions. In: Springer. *Computer Aided Verification: 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings 26.* 2014, p. 646–662.

[67] LIANG, T., REYNOLDS, A., TSISKARIDZE, N., TINELLI, C., BARRETT, C. and DETERS, M. An efficient SMT solver for string constraints. *Formal Methods in System Design*. Springer. 2016, vol. 48, no. 3, p. 206–234. ISSN 0925-9856.

[68] LIANG, T., TSISKARIDZE, N., REYNOLDS, A., TINELLI, C. and BARRETT, C. A decision procedure for regular membership and length constraints over unbounded strings. In: Springer. *International Symposium on Frontiers of Combining Systems*. 2015, p. 135–150.

[69] LIN, A. W. and BARCELÓ, P. String solving with word equations and transducers: towards a logic for analysing mutation XSS. In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Association for Computing Machinery, 2016, p. 123–136. DOI: 10.1145/2837614.2837641. ISBN 9781450335492. Available at: https://doi.org/10.1145/2837614.2837641.

[70] MAKANIN, G. THE PROBLEM OF SOLVABILITY OF EQUATIONS IN A FREE SEMIGROUP. *Mathematics of the USSR-Sbornik*. 1977, vol. 32, no. 2.

[71] MCMILLAN, K. L. Interpolation and SAT-Based Model Checking. In: HUNT, W. A. and SOMENZI, F., ed. *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*. 2003, p. 1–13. DOI: 10.1007/978-3-540-45069-6_1. ISBN 978-3-540-45069-6. Available at: http://dx.doi.org/10.1007/978-3-540-45069-6_1.

[72] MORVAN, C. On Rational Graphs. In: TIURYN, J., ed. *Foundations of Software Science and Computation Structures*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, p. 252–266. ISBN 978-3-540-46432-7.

[73] NIEUWENHUIS, R., OLIVERAS, A. and TINELLI, C. Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL (T). *Journal of the ACM (JACM)*. ACM New York, NY, USA. 2006, vol. 53, no. 6, p. 937–977. DOI: 10.1145/1217856.1217859. ISSN 0004-5411. Available at: https://doi.org/10.1145/1217856.1217859.

[74] NÖTZLI, A., REYNOLDS, A., BARBOSA, H., BARRETT, C. and TINELLI, C. Even faster conflicts and lazier reductions for string solvers. In: SHOHAM, S. and VIZEL, Y., ed. *International Conference on Computer Aided Verification*. 2022, p. 205–226. ISBN 978-3-031-13188-2.

[75] OWASP. *Top 10* [https://www.owasp.org/images/f/f8/OWASP_Top_10_-_2013.pdf]. 2013.

[76] OWASP. *Top 10* [https://owasp.org/www-project-top-ten/2017/]. 2017.

[77] OWASP. *Top 10* [https://owasp.org/Top10/]. 2021.

[78] PLANDOWSKI, W. Satisfiability of word equations with constants is in PSPACE. In: *40th Annual Symposium on Foundations of Computer Science (Cat. No.99CB37039)*. 1999, p. 495–500. DOI: 10.1109/SFFCS.1999.814622.

[79] PLANDOWSKI, W. Satisfiability of word equations with constants is in PSPACE. *J. ACM*. 2004, vol. 51, no. 3, p. 483–496. ISSN 0004-5411.

[80] Plandowski, W. An efficient algorithm for solving word equations. In: *STOC*. 2006, p. 467–476.

[81] Quine, W. V. Concatenation as a basis for arithmetic. *J. Symb. Log.* 1946, vol. 11, no. 4.

[82] Reynolds, A., Nötzli, A., Barrett, C. W. and Tinelli, C. Reductions for Strings and Regular Expressions Revisited. In: *FMCAD*. 2020, p. 225–235. DOI: 10.34727/2020/isbn.978-3-85448-042-6_30.

[83] Reynolds, A., Woo, M., Barrett, C., Brumley, D., Liang, T. and Tinelli, C. Scaling up DPLL (T) string solvers using context-dependent simplification. In: Springer. *International Conference on Computer Aided Verification*. 2017, p. 453–474.

[84] Robson, J. M. and Diekert, V. On quadratic word equations. In: Springer. *STACS 99: 16th Annual Symposium on Theoretical Aspects of Computer Science Trier, Germany, March 4–6, 1999 Proceedings 16*. 1999, p. 217–226.

[85] Rümmer, P. 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*. Spv, 2008, vol. 5330, p. 274–289. LNCS. ISBN 978-3-540-89438-4.

[86] Sakarovitch, J. *Elements of automata theory*. Cambridge University Press, 2009.

[87] Saxena, P., Akhawe, D., Hanna, S., Mao, F., McCamant, S. and Song, D. A symbolic execution framework for javascript. In: IEEE. *2010 IEEE Symposium on Security and Privacy*. 2010, p. 513–528.

[88] Schulz, K. U. Makanin's algorithm for word equations-two improvements and a generalization. In: Springer. *Word Equations and Related Topics*. 1992, p. 85–150. ISBN 978-3-540-46737-3.

[89] Scott, J. D., Flener, P., Pearson, J. and Schulte, C. Design and Implementation of Bounded-Length Sequence Variables. In: Salvagnin, D. and Lombardi, M., ed. *Integration of AI and OR Techniques in Constraint Programming*. Cham: Springer International Publishing, 2017, p. 51–67. ISBN 978-3-319-59776-8.

[90] Security, H. N. *Analysis of three billion attacks reveals SQL injections cost $196,000* [https://www.helpnetsecurity.com/2014/03/28/analysis-of-three-billion-attacks-reveals-sql-injections-cost-196000/]. 2014.

[91] Sen, K., Kalasapur, S., Brutch, T. and Gibbs, S. Jalangi: A selective record-replay and dynamic analysis framework for JavaScript. In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. 2013, p. 488–498.

[92] Sen, K., Marinov, D. and Agha, G. CUTE: A concolic unit testing engine for C. *ACM SIGSOFT Software Engineering Notes*. ACM New York, NY, USA. 2005, vol. 30, no. 5, p. 263–272.

[93] SHEERAN, M., SINGH, S. and STÅLMARCK, G. Checking Safety Properties Using Induction and a SAT-Solver. In: *FMCAD*. Springer, 2000, vol. 1954, p. 108–125. LNCS. ISBN 3-540-41219-0.

[94] STANFORD, C., VEANES, M. and BJØRNER, N. Symbolic Boolean derivatives for efficiently solving extended regular expression constraints. In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 2021, p. 620–635.

[95] TECHNICA, A. *Feds: Sailor hacked Navy network while aboard nuclear aircraft carrier* [https://arstechnica.com/information-technology/2014/05/feds-sailor-hacked-navy-network-while-aboard-nuclear-aircraft-carrier/]. 2014.

[96] TRINH, M.-T., CHU, D.-H. and JAFFAR, J. S3: A symbolic string solver for vulnerability detection in web applications. In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. 2014, p. 1232–1243.

[97] TRINH, M.-T., CHU, D.-H. and JAFFAR, J. Progressive reasoning over recursively-defined strings. In: Springer. *Computer Aided Verification: 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I 28*. 2016, p. 218–240.

[98] TWISTIT.TECH. *PHP Tutorials* [https://www.makephpsites.com/php-tutorials/user-management-tools/changing-passwords.php]. 2019. [Online; accessed 2019-04-29]. Available at: https://www.makephpsites.com/php-tutorials/user-management-tools/changing-passwords.php.

[99] VARDI, M. Y. 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)*. 1995, p. 238–266. DOI: 10.1007/3-540-60915-6_6. Available at: http://dx.doi.org/10.1007/3-540-60915-6_6.

[100] VEANES, M., HOOIMEIJER, P., LIVSHITS, B., MOLNAR, D. and BJØRNER, N. Symbolic finite state transducers: Algorithms and applications. In: *POPL'12*. ACM Trans. Comput. Log., 2012, p. 137–150.

[101] VERMA, K. N., SEIDL, H. and SCHWENTICK, T. On the Complexity of Equational Horn Clauses. In: *CADE'05*. 2005, p. 337–352.

[102] WANG, H.-E., CHEN, S.-Y., YU, F. and JIANG, J.-H. R. A Symbolic Model Checking Approach to the Analysis of String and Length Constraints. In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2018, p. 623–633. ASE 2018. DOI: 10.1145/3238147.3238189. ISBN 9781450359375. Available at: https://doi.org/10.1145/3238147.3238189.

[103] WANG, H., TSAI, T., LIN, C., YU, F. and JIANG, J. R. 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. Springer, 2016, vol. 9779, p. 241–260. Lecture Notes in Computer Science. DOI: 10.1007/978-3-319-41528-4. ISBN 978-3-319-41527-7. Available at: http://dx.doi.org/10.1007/978-3-319-41528-4.

[104] Weinberger, J., Saxena, P., Akhawe, D., Finifter, M., Shin, R. and Song, D. A systematic analysis of XSS sanitization in web application frameworks. In: Springer. *Computer Security–ESORICS 2011: 16th European Symposium on Research in Computer Security, Leuven, Belgium, September 12-14, 2011. Proceedings 16*. 2011, p. 150–171.

[105] Yu, F., Alkhalaf, M. and Bultan, T. Stranger: An automata-based string analysis tool for php. In: Springer. *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 2010, p. 154–157.

[106] Yu, F., Alkhalaf, M., Bultan, T. and Ibarra, O. H. Automata-based symbolic string analysis for vulnerability detection. *Formal Methods in System Design*. Springer. 2014, vol. 44, p. 44–70.

[107] Yu, F., Bultan, T. and Ibarra, O. H. Relational string verification using multi-track automata. *International Journal of Foundations of Computer Science*. World Scientific. 2011, vol. 22, no. 08, p. 1909–1924.

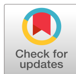
[108] Zheng, Y., Ganesh, V., Subramanian, S., Tripp, O., Dolby, J. and Zhang, X. Effective search-space pruning for solvers of string equations, regular expressions and length constraints. In: Springer. *Computer Aided Verification: 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I 27*. 2015, p. 235–254.

[109] Zheng, Y., Zhang, X. and Ganesh, V. Z3-str: A z3-based string solver for web application analysis. In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. 2013, p. 114–124.

Appendix A

Papers

The following papers form the main part of this thesis.



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;

4

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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2018 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.
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; Jerry')">Tom &amp; Jerry</button>
```

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

```
<button onclick="viewPerson('&#39;);attackScript();//')">&#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 $q \rightarrow_\psi \varphi$ 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* $\iota : \text{var}(\varphi) \to \Sigma^*$ 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 $|\text{free}(\psi) \cap \text{free}(\psi')| \leq 1$, 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 $x = \mathcal{R}_1(\mathsf{name}) \land y = \mathcal{R}_2(x) \land z = w_1 \circ y \circ w_2 \circ x \circ w_3 \land u = \mathcal{R}_3(z)$. 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 $\neg v_1 \land \neg v_0$, b as $\neg v_1 \land v_0$, c as $v_1 \land \neg v_0$, and d as $v_1 \land v_0$. 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 $w = b_1 \dots b_m$, $m \geq 0$, 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 $\rho = \rho_0 b_1 \rho_1 \dots b_m \rho_m$ 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 $\varphi_{\mathrm{Pres}}(|x_1|, \ldots, |x_k|)$, where φ_{Pres} is a Presburger formula.

Definition 5.5 (Extended acyclic formulae). A string formula φ augmented with length constraints $\varphi_{\mathrm{Pres}}(|x_1|, \ldots, |x_k|)$ 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 $|\mathrm{free}(\psi) \cap \mathrm{free}(\psi')| \leq 1$, 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 $\Delta(q_n) = \mathtt{false}$. 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 $\varphi(x, y) \land \psi(x, y)$ where one of the conjuncts, say φ, can be rewritten as $\varphi_1[\bar{s}_1](x) \land \varphi_2[\bar{s}_2](y)$. The whole formula can be written as $\varphi_1[\bar{s}_1](x) \land (\varphi_2[\bar{s}_2](y) \land \psi(x, y))$, 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 $\mathcal{R}_1[\bar{s}](x) \land \mathcal{R}_2[\bar{s}](y)$ 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 $\square \in \{\land, \lor\}$, $\mathcal{R}_\varphi[\bar{r}](\bar{x})$, and $\mathcal{R}_\psi[\bar{s}](\bar{y})$, the constraint $\mathcal{R}_{\varphi\square\psi}[\bar{t}](\bar{z})$ 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 $\square$ unchanged, and $\{\bar{t}\} = \{\bar{r}\} \cup \{\bar{s}\}$.

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

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 $q_2 \land q_3$. 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).
- $I' = I^+ \land \mathit{Choose}$ 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 $\mathit{Choose} = \bigwedge_{i=1}^{n} s_i^+ \lor s_i^-$ enforces that every parameter has a value.
- $F' = F^- \land \mathit{Disambiguate}$ 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 $\mathit{Disambiguate} = \bigwedge_{i=1}^{n} \neg s_i^+ \lor \neg s_i^-$ 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 $\varphi : y = \mathcal{R}(x) \land z = x \circ y \land \mathcal{A}(z)$. The AFT $\mathcal{R} = (W\langle 2\rangle, Q = \{q\}, \Delta = \{q \mapsto q \land \neg \mathsf{d}^1 \land (\mathsf{c}^1 \to \mathsf{d}^2) \land \bigwedge_{a \in \Sigma\setminus\{\mathsf{c}\}}(a^1 \leftrightarrow a^2))\}, I = q, F = \mathtt{true})$ 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 $\mathcal{A} = (V, Q' = \{r_0, r_1, r_2\}, \Delta', I' = r_0, F' = \neg r_0 \land \neg r_1)$ where $\Delta'(r_0) = (r_0 \land \neg \mathsf{c}) \lor (r_1 \land \mathsf{c})$, $\Delta'(r_1) = (r_1 \land \neg \mathsf{d}) \lor (r_2 \land \mathsf{d})$, 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 $\psi \land \bigwedge_{i=1}^{m} x_i = F_i$ 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 $P(x) \land \psi$ is equivalent to $\exists x' : x' = \mathcal{R}(x) \land \psi$ 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 $\varphi_0' : y = \mathcal{R}(x) \land z = x \circ y \land z' = \mathcal{S}(z)$ 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 $I' \wedge \bigwedge_{i=0}^{2}(r_i \rightarrow s_0)$ and $\mathcal{S}_2$ has the final formula $F' \wedge \bigwedge_{i=0}^{2}(s_i \rightarrow r_i)$. 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 $x \rightarrow y$ 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:

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

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 $\bar{q}'' \geq \bar{q}'$, writing $\bar{p} \leq \bar{q}$ for the point-wise order on bit-vectors $\bar{p}, \bar{q} \in \mathbb{B}^m$ (i.e., $\bar{p} \leq \bar{q}$ 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 $\mathit{Final}^{di}[\bar{q}]$ 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 $\preceq$-maximal states (or $\preceq$-minimal states) of the transition system when checking reachability of $\mathit{Final}^{di}[\bar{q}]$. 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 $T^{\min}_{\mathcal{A}} = (\mathbb{B}^m, \mathit{Init}^{\min}, \mathit{Trans}^{\min})$ obtained in this way can abstractly be defined as:

$$\mathit{Init}^{\min}[\bar{q}] = \mathit{Init}^{di}[\bar{q}] \wedge \forall \bar{p} \prec \bar{q}.\ \neg \mathit{Init}^{di}[\bar{p}] \quad (2)$$
$$\mathit{Trans}^{\min}[\bar{q}, \bar{q}'] = \mathit{Trans}^{di}[\bar{q}, \bar{q}'] \wedge \forall \bar{p} \prec \bar{q}'.\ \neg \mathit{Trans}^{di}[\bar{q}, \bar{p}] \quad (3)$$

That means, $\mathit{Init}^{\min}$ and $\mathit{Trans}^{\min}$ are defined to only retain the $\preceq$-minimal states. Computing $\mathit{Init}^{\min}$ and $\mathit{Trans}^{\min}$ 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 $T^{\mathrm{im}}_{\mathcal{A}} = (\mathbb{B}^m, \mathit{Init}^{\mathrm{im}}, \mathit{Trans}^{\mathrm{im}})$ by only minimising post-states in $\mathit{Trans}^{\mathrm{im}}$ with respect to the same input letter V_n:

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

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 $\varphi[\bar{q}] \wedge \bigwedge_{i=0}^{m-1} q_i \to \neg\varphi[\bar{q}][q_i/\mathtt{false}]$, where we write $\varphi[q_i/\mathtt{false}]$ 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 $Init^{\mathrm{im}}$ and $Trans^{\mathrm{im}}$ is:

$$Init^{\mathrm{im}}[\bar{q}] \equiv Init^{di}[\bar{q}] \wedge \bigwedge_{i=0}^{m-1} q_i \to \neg Init^{di}[\bar{q}][q_i/\mathtt{false}] \tag{4}$$

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

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 $Init^{di}[\bar{q}][q_i/\mathtt{false}]$ and $Trans[\bar{q},\bar{q}',V_n][q_i'/\mathtt{false}]$ 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 $\Delta(q_4) = \mathtt{false}$.

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

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

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):

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

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

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 $Trans^{im}[\bar{q}, \bar{q}']$:

$$Init^{det}[\bar{q}] = q_0 \wedge \neg q_1 \wedge \neg q_2 \wedge \neg q_3 \wedge \neg q_4$$

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

□

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:

$$Init^{\mathrm{det}}[\bar{q}] = q_0 \wedge \bigwedge_{i=1}^{m-1} \neg q_i$$

$$Trans^{\mathrm{det}}[\bar{q}, \bar{q}'] = \exists H. \left(\left(\bigwedge_{i=0}^{m-1} q_i' \leftrightarrow NewState(q_i) \right) \wedge \exists V_n. \left(\bigwedge_{i=0}^{m-1} q_i \to InputInv(\Delta(q_i), i) \right) \right)$$

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:

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

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

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

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.

Chain-Free String Constraints

Parosh Aziz Abdulla[1], Mohamed Faouzi Atig[1], Bui Phi Diep[1](✉), Lukáš Holík[2], and Petr Janků[2]

[1] Uppsala University, Uppsala, Sweden
{parosh,mohamed_faouzi.atig,bui.phi-diep}@it.uu.se
[2] Brno University of Technology, Brno, Czech Republic
{holik,ijanku}@fit.vutbr.cz

Abstract. We address the satisfiability problem for string constraints that combine relational constraints represented by transducers, word equations, and string length constraints. This problem is undecidable in general. Therefore, we propose a new decidable fragment of string constraints, called weakly chaining string constraints, for which we show that the satisfiability problem is decidable. This fragment pushes the borders of decidability of string constraints by generalising the existing straight-line as well as the acyclic fragment of the string logic. We have developed a prototype implementation of our new decision procedure, and integrated it into in an existing framework that uses CEGAR with under-approximation of string constraints based on flattening. Our experimental results show the competitiveness and accuracy of the new framework.

Keywords: String constraints · Satisfiability modulo theories · Program verification

1 Introduction

The recent years have seen many works dedicated to extensions of SMT solvers with new background theories that can lead to efficient analysis of programs with high-level data types. A data type that has attracted a lot of attention is *string* (for instance [2,4,7,9,10,14,16–18,20,33,37,38]). Strings are present in almost all programming and scripting languages. String solvers can be extremely useful in applications such as verification of string-manipulating programs [4] and analysis of security vulnerabilities of scripting languages (e.g., [20,29,30,37]). The wide range of the commonly used primitives for manipulating strings in such languages requires string solvers to handle an expressive class of string logics. The most important features that a string solver have to model are *concatenation* (which is used to express assignments in programs), *transduction* (which can be used to model sanitisation and replacement operations), and *string length* (which is used to constraint lengths of strings).

This work has been supported by the Czech Science Foundation (project No. 19-24397S), the IT4Innovations Excellence in Science (project No. LQ1602), and the FIT BUT internal projects FIT-S-17-4014 and FEKT/FIT-J-19-5906.

© Springer Nature Switzerland AG 2019
Y.-F. Chen et al. (Eds.): ATVA 2019, LNCS 11781, pp. 277–293, 2019.
https://doi.org/10.1007/978-3-030-31784-3_16

It is well known that the satisfiability problem for the full class of string constraints with concatenation, transduction, and length constraints is undecidable in general [10, 23] even for a simple formula of the form $\mathcal{T}(x,x)$ where $\mathcal{T}$ is a rational transducer and x is a string variable. However, this theoretical barrier did not prevent the development of numerous efficient solvers such as Z3-str3 [7], Z3-str2 [38], CVC4 [18], S3P [33, 34], and TRAU [2,3]. These tools implement semi-algorithms to handle a large variety of string constraints, but do not provide completeness guarantees. Another direction of research is to find meaningful and expressive subclasses of string logics for which the satisfiability problem is decidable. Such classes include the acyclic fragment of Norn [5], the solved form fragment [13], and also the straight-line fragment [9,14,20].

In this paper, we propose an approach which is a mixture of the two above research directions, namely finding decidable fragments and making use of it to develop efficient semi-algorithms. To that aim, we define the class of *chain-free* formulas which strictly subsumes the acyclic fragment of Norn [5] as well as the straight-line fragment of [9,14,20], and thus extends the known border of decidability for string constraints. The extension is of a practical relevance. A straight-line constraint models a path through a string program in the single static assignment form, but as soon as the program compares two initialised string variables, the string constraint falls out of the fragment. The acyclic restriction of Norn on the other hand does not include transducer constraints and does not allow multiple occurrences of a variable in a single string constraint (e.g. an equation of the form $xy = zz$). Our chain-free fragment is liberal enough to accommodate constraints that share both these forbidden features (including $xy = zz$).

The following pseudo-PHP code (a variation of a code at [35]) that prompts a user to change his password is an example of a program that generates a chain-free constraint that is neither straight-line nor acyclic according to [4,20].

```
$old=$database->real_escape_string($oldIn);
$new=$database->real_escape_string($newIn);
$pass=$database->query("SELECT password FROM users WHERE userID=".$user);
if($pass == $old)
   if($new != $old)
      $query = "UPDATE users SET password=".$new." WHERE userID=".$user;
      $database->query($query);
```

The user inputs the old password `oldIn` and the new password `newIn`, both are sanitized and assigned to `old` and `new`, respectively. The old sanitized password is compared with the value `pass` from the database, to authenticate the user, and then also with the new sanitized password, to ensure that a different password was chosen, and finally saved in the database. The sanitization is present to prevent SQL injection. To ensure that the sanitization works, we wish to verify that the SQL query `query` is safe, that is, it does not belong to a regular language *Bad* of dangerous inputs. This safety condition is expressed by the constraint

$$\texttt{new} = \mathcal{T}(\texttt{newIn}) \wedge \texttt{old} = \mathcal{T}(\texttt{oldIn}) \wedge \texttt{pass} = \texttt{old} \wedge \texttt{new} \neq \texttt{old} \\ \wedge\, \texttt{query} = u.\texttt{new}.v.\texttt{user} \wedge \texttt{query} \in \mathit{Bad}$$

The sanitization on lines 1 and 2 is modeled by the transducer $\mathcal{T}$, and u and v are the constant strings from line 7. The constraints fall out from the straight-line due to the test

new $\neq$ old. The main idea behind the chain-free fragment is to associate to the set of relational constraints a *splitting graph* where each node corresponds to an occurrence of a variable in the relational constraints of the formula (as shown in Fig. 1). An edge from an occurrence of x to an occurrence of y means that the source occurrence of x appears in a relational constraint which has in the opposite side an occurrence of y different from the target occurrence of y. The chain-free fragment prohibits loops in the graph, that we call *chains*, such as those shown in red in Fig. 1.

Then, we identify the so called *weakly chaining* fragment which strictly extends the chain-free fragment by allowing *benign* chains. Benign chains relate relational constraints where each left side contains only one variable, the constraints are all *length preserving*, and all the nodes of the cycles appear exclusively on the left or exclusively on the right sides of the involved relational constraints (as is the case in Fig. 1). Weakly chaining constraints may in practice arise from the checking that an encoding followed a decoding function is indeed the identity, i.e., satisfiability of constraints of the form $\mathcal{T}_{\mathrm{enc}}(\mathcal{T}_{\mathrm{dec}}(x)) = x$, discussed e.g. in [15]. For instance, in situations similar to the example above, one might like to verify that the sanitization of a password followed by the application of a function supposed to invert the sanitization gives the original password.

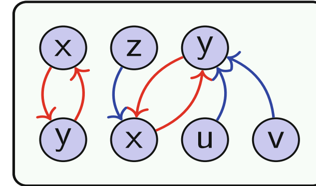


Fig. 1. The splitting graph of $x = z \cdot y \wedge y = x \cdot u \cdot v$. (Color figure online)

Our decision procedure for the weakly chaining formulas proceeds in several steps. The formula is transformed to an equisatisfiable chain-free formula, and then to an equisatisfiable concatenation free formula in which the relational constraints are of the form $\mathcal{T}(x,y)$ where x and y are two string variables and $\mathcal{T}$ is a transducer/relational constraint. Finally, we provide a decision procedure of a chain and concatenation-free formulae. The algorithm is based on two techniques. First, we show that the chain-free conjunction over relational constraints can be turned into a single equivalent transducer constraint (in a similar manner as in [6]). Second, consistency of the resulting transducer constraint with the input length constraints is checked via the computation of the Parikh image of the transducer.

To demonstrate the usefulness of our approach, we have implemented our decision procedure in SLOTH [14], and then integrated it in the open-source solver TRAU [2,3]. TRAU is a string solver which is based on a Counter-Example Guided Abstraction Refinement (CEGAR) framework which contains both an under- and an over-approximation module. These two modules interact together in order to automatically make these approximations more precise. We have implemented our decision procedure inside the over-approximation module which takes as an input a constraint and checks if it belongs to the weakly chaining fragment. If it is the case, then we use our decision procedure outlined above. Otherwise, we start by choosing a minimal set of occurrences of variables x that needs to be replaced by fresh ones such that the resulting constraint falls in our decidable fragment. We compare our prototype implementation against four other state-of-the-art string solvers, namely Ostrich [10], Z3-str3 [7], CVC4 [18,19], and TRAU [1]. For our comparison with Z3-str3, we use the version that is part of Z3

4.8.4. Our experimental results show the competitiveness as as well as accuracy of the framework compared to the solver TRAU [2,3]. Furthermore, the experimental results show the competitiveness and generality of our method compared to the existing techniques. In summary, our main contributions are: (1) a new decidable fragment of string constraints, called chain-free, which strictly generalises the existing straight-line as well as the acyclic fragment [4,20] and precisely characterises the decidability limitations of general relational/transducer constraints combined with concatenation, (2) a relaxation of the chain-free fragment, called weakly chaining, which allows special chains with length preserving relational constraints, (3) a decision procedures for checking the satisfiability problem of chain-free as well as weakly chaining string constraints, and (4) a prototype with experimental results that demonstrate the efficiency and generality of our technique on benchmarks from the literature as well as on new benchmarks.

2 Preliminaries

Sets and Strings. We use $\mathbb{N}$, $\mathbb{Z}$ to denote the sets of natural numbers and integers, respectively. A finite set Σ of *letters* is an *alphabet*, a sequence of symbols $a_1 \cdots a_n$ from Σ is a *word* or a *string* over Σ, with its *length* n denoted by $|w|$, ε is the *empty word* with $|\varepsilon| = 0$, it is a neutral element with respect to string concatenation $\circ$, and Σ^* is the set of all words over Σ including ε.

Logic. Given a predicate formula, an occurrence of a predicate is *positive* if it is under an even number of negations. A formula is in *disjunctive normal form* (DNF) if it is a disjunction of *clauses* that are themselves conjunctions of (negated) predicates. We write $\Psi[x/t]$ to denote the formula obtained by substituting in the formula Ψ each occurrence of the variable x by the term t.

(Multi-tape)-Automata and Transducers. A *Finite Automaton* (FA) over an alphabet Σ is a tuple $\mathcal{A} = \langle Q, \Delta, I, F \rangle$, where Q is a finite set of *states*, $\Delta \subseteq Q \times \Sigma_\varepsilon \times Q$ with $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$ is a set of *transitions*, and $I \subseteq Q$ (resp. $F \subseteq Q$) are the *initial* (resp. *accepting*) states. $\mathcal{A}$ accepts a word w iff there is a sequence $q_0 a_1 q_1 a_2 \cdots a_n q_n$ such that $(q_{i-1}, a_i, q_i) \in \Delta$ for all $1 \leq i \leq n$, $q_0 \in I$, $q_n \in F$, and $w = a_1 \circ \cdots \circ a_n$. The *language* of $\mathcal{A}$, denoted $\mathcal{L}(\mathcal{A})$, is the set all accepted words.

Given $n \in \mathbb{N}$, a *n-tape automaton* $\mathcal{T}$ is an automaton over the alphabet $(\Sigma_\varepsilon)^n$. It *recognizes* the relation $\mathcal{R}(\mathcal{T}) \subseteq (\Sigma^*)^n$ that contains vectors of words $(w_1, w_2, \ldots, w_n)$ for which there is $(a_{(1,1)}, a_{(2,1)}, \ldots, a_{(n,1)}) \cdots (a_{(1,m)}, a_{(2,m)}, \ldots, a_{(n,m)}) \in \mathcal{L}(\mathcal{T})$ with $w_i = a_{(i,1)} \circ \cdots \circ a_{(i,m)}$ for all $i \in \{1, \ldots, n\}$. A n-tape automaton $\mathcal{T}$ is said to be *length-preserving* if its transition relation $\Delta \subseteq Q \times \Sigma^n \times Q$. A *transducer* is a 2-tape automaton.

Let us recall some well-know facts about the class of multi-tape automata. First, the class of n-tape automata is closed under union but not under complementation nor intersection. However, the class of *length-preserving* multi-tape automata is closed under intersection. Multi-tape automata are closed under composition. Let $\mathcal{T}$ and $\mathcal{T}'$ be two multi-tape automata of dimension n and m, respectively, and let $i \in \{1, \ldots, n\}$ and $j \in \{1, \ldots, m\}$ be two indices. Then, it is possible to construct a $(n+m-1)$-tape automaton $\mathcal{T} \wedge_{(i,j)} \mathcal{T}'$ which accepts the set of words $(w_1, \ldots, w_n, u_1, \ldots, u_{j-1}, u_{j+1}, \ldots, u_m)$ if and

only if $(w_1,\dots,w_n) \in \mathcal{R}(\mathcal{T})$ and $(u_1,\dots,u_{j-1},w_i,u_{j+1},\dots,u_m) \in \mathcal{R}(\mathcal{T}')$. Furthermore, we can show that multi-tape automata are closed under permutations: Given a permutation $\sigma : \{1,\dots,n\} \to \{1,\dots,n\}$ and a n-tape automaton $\mathcal{T}$, it is possible to construct a n-tape automaton $\sigma(\mathcal{T})$ such that $\mathcal{R}(\sigma(\mathcal{T})) = \{(w_{\sigma(1)},\dots,w_{\sigma(n)}) \mid (w_1,w_2,\dots,w_n) \in \mathcal{R}(\mathcal{T})\}$. Finally, given a n-tape automaton $\mathcal{T}$ and a natural number $k \geq n$, we can construct a k-tape automaton s. t. $(w_1,\dots,w_k) \in \mathcal{R}(\mathcal{T}')$ if and only if $(w_1,\dots,w_n) \in \mathcal{R}(\mathcal{T})$.

3 String Constraints

The syntax of a string formula Ψ over an alphabet Σ and a set of variables $\mathbb{X}$ is given in Fig. 2. It is a Boolean combination of memberships, relational, and arithmetic constraints over string terms t_{str} (i.e., concatenations of variables in $\mathbb{X}$). *Membership constraints* denote membership in the language of a finite-state automaton $\mathcal{A}$ over Σ. *Relational constraints* denote either an equality of string terms, which we normally write as $t = t'$ instead of $=(t,t')$, or that the terms are related by a relation recognised by a transducer $\mathcal{T}$. (Observe that the equality relations can be also expressed using length preserving transducers.) Finally, arithmetic terms t_{ar} are linear functions over term lengths and integers, and arithmetic constraints are inequalities of arithmetic terms. String formulae allow using negation with one restriction, namely, constraints that are *not invertible* must have only positive occurrences. General transducers are not invertible, it is not possible to negate them. Regular membership, length preserving relations (including equality), and length constraints are invertible.

$$
\begin{aligned}
\Psi &::= \varphi \mid \Psi \wedge \Psi \mid \Psi \vee \Psi \mid \neg\Psi \\
\varphi &::= \mathcal{A}(t_{str}) \mid R(t_{str},t_{str}) \mid t_{ar} \geq t_{ar} \\
R &::= \mathcal{T} \mid = \\
t_{str} &::= \varepsilon \mid x \mid t_{str} \circ t_{str} \\
t_{ar} &::= k \mid |t_{str}| \mid t_{ar} + t_{ar}
\end{aligned}
$$

Fig. 2. Syntax of string formulae

To simplify presentation, we do not consider *mixed* string terms t_{str} that contain, besides variables of $\mathbb{X}$, also symbols of Σ. This is without loss of generality because a mixed term can be encoded as a conjunction of the pure term over $\mathbb{X}$ obtained by replacing every occurrence of a letter $a \in \Sigma$ by a fresh variable x and the regular membership constraints $\mathcal{A}_a(x)$ with $\mathcal{L}(\mathcal{A}_a) = \{a\}$. Observe also that membership and equality constraints may be expressed using transducers.

Semantics. We describe the semantics of our logic using a mapping η, called *interpretation*, that assigns to each string variable in $\mathbb{X}$ a word in Σ^*. Extended to string terms by $\eta(t_{s_1} \circ t_{s_2}) = \eta(t_{s_1}) \circ \eta(t_{s_2})$. Extended to arithmetic terms by $\eta(|t_s|) = |\eta(t_s)|$, $\eta(k) = k$ and $\eta(t_i + t_i') = \eta(t_i) + \eta(t_i')$. Extended to atomic constraints, η returns a truth value:

$$
\begin{aligned}
\eta(\mathcal{A}(t_{str})) = \top &\quad \text{iff} \quad \eta(t_{str}) \in \mathcal{L}(\mathcal{A}) \\
\eta(R(t_{str},t'_{str})) = \top &\quad \text{iff} \quad (\eta(t_{str}),\eta(t'_{str})) \in \mathcal{R}(R) \\
\eta(t_{i_1} \leq t_{i_2}) = \top &\quad \text{iff} \quad \eta(t_{i_1}) \leq \eta(t_{i_2})
\end{aligned}
$$

Given two interpretations η_1 and η_2 over two disjoint sets of string variables $\mathbb{X}_1$ and $\mathbb{X}_2$, respectively. We use $\eta_1 \cup \eta_2$ to denote the interpretation over $\mathbb{X}_1 \cup \mathbb{X}_2$ such that $(\eta_1 \cup \eta_2)(x) = \eta_1(x)$ if $x \in \mathbb{X}_1$ and $(\eta_1 \cup \eta_2)(x) = \eta_2(x)$ if $x \in \mathbb{X}_2$.

The truth value of a Boolean combination of formulae under η is defined as usual. If $\eta(\Psi) = \top$ then η is a *solution* of Ψ, written $\eta \models \Psi$. The formula Ψ is *satisfiable* iff it has a solution, otherwise it is *unsatisfiable*.

A relational constraint is said to be *left-sided* if and only if it is on the form $R(x, t_{str})$ where $x \in \mathbb{X}$ is a string variable and t_{str} is a string term. Any string formula can be transformed into a formula where all the relational constraints are left-sided by replacing any relational constraint of the form $R(t_{str}, t'_{str})$ by $R(x, t'_{str}) \wedge x = t$ where x is fresh.

A formula Ψ is said to be *concatenation free* if and only if for every relational constraint $R(t_{str}, t'_{str})$, the string terms t_{str} and t'_{str} appearing in the parameters of any relational constraints in Ψ are variables (i.e., $t_{str}, t'_{str} \in \mathbb{X}$).

4 Chain Free and Weakly Chaining Fragment

It is well known that the satisfiability problem for the class of string constraint formulas is undecidable in general [10,23]. This problem is undecidable already for a single transducer constraint of the form $\mathcal{T}(x,x)$ (by a simple reduction from the Post-Correspondence Problem). In the following, we define a subclass called *weakly chaining fragment* for which we prove that the satisfiability problem is decidable.

Splitting Graph. Let $\Psi ::= \bigwedge_{j=1}^{m} \varphi_j$ be a conjunction of relational string constraints with $\varphi_j ::= R_j(t_{2j-1}, t_{2j}), 1 \leq j \leq m$ where for each $i : 1 \leq i \leq 2m$, t_i is a concatenation of variables $x_i^1 \circ \cdots \circ x_i^{n_i}$. We define the set of *positions* of Ψ as $P = \{(i, j) \mid 1 \leq j \leq 2m \;\wedge\; 1 \leq i \leq n_j\}$. The *splitting graph* of Ψ is then the graph $G_\Psi = (P, E, \mathtt{var}, \mathtt{con})$ where the positions in P are its nodes, and the mapping $\mathtt{var} : P \to \mathbb{X}$ labels each position (i, j) with the variable x_j^i appearing at that position. We say that $(i, 2j-1)$ (resp. $(i, 2j)$) is the ith *left* (resp. *right*) positions of the jth constraint, and that R_j is the predicate of these positions. Any pair of a left and a right position of the same constraint are called *opposing*. The set of edges E then consists of edges (p, p') between positions for which there is an intermediate position p'' (different from p') that is opposing to p and is labeled by the same variable as p' ($\mathtt{var}(p'') = \mathtt{var}(p')$). Finally, the labelling $\mathtt{con}$ of edges assigns to (p, p') the constraint of p, that is, $\mathtt{con}(p, p') = R_j$ where p is a position of the jth constraint. An example of a splitting graph is on Fig. 1.

Chains. A *chain*[1] in the graph is a sequence of the form $(p_0, p_1), (p_1, p_2), \ldots, (p_n, p_0)$ of edges in E. A chain is *benign* if (1) all the relational constraints corresponding to the edges $\mathtt{con}(p_0, p_1), \mathtt{con}(p_1, p_2), \ldots, \mathtt{con}(p_n, p_0)$ are left sided and and all the string relations involved in these constraints are length preserving, and (2) the sequence of positions $p_0, p_1, \ldots, p_n$ consists of left positions only, or from right positions only. Observe that if there is a benign chain that uses only right positions then there exists also a benign chain that uses only left positions. The graph is *chain-free* if it has no chains, and it is *weakly chaining* if all its chains are benign. A formula is *chain-free* (resp. *weakly chaining*) if the splitting graph of every clause in its DNF is chain-free (resp. weakly chaining). Benign chains are on Fig. 1 shown in red.

[1] We use chains instead of cycles in order to avoid confusion between our decidable fragment and the ones that exist in literatures.

In the following sections, we will show decision procedures for the chain-free and weakly chaining fragments. Particularly, we will show how a weakly chaining formula can be transformed to a chain-free formula by elimination of benign cycles, how then concatenation can be eliminated from a chain-free formula, and finally how to decide a concatenation free-formula.

Undecidability of Chaining Formulae. Before presenting the decision procedures for weakly chaining formulae, we finish the current section by stating that the chain-free fragment is indeed the limit of decidability of general transducer constraints, in the following sense: We say that two conjunctive string formulae have the same *relation-concatenation skeleton* if one can be obtained from the other by removing membership and length constraints and replacing a constraint of the form $R(t,t')$ by another constraint of the form $R'(t,t')$. A *skeleton class* is then an equivalence class of string formulae that have the same relation-concatenation skeleton.

Lemma 1. *The satisfiability problem is undecidable for every given* skeleton class.

The proof of the above lemma can be done through a reduction from undecidability of general transducer constraints of the form $\mathcal{T}(x,x)$. Together with decidability of chain-free formulae, discussed in Sects. 6 and 7, the lemma implies that the satisfiability problem for a skeleton class is decidable *if and only if* its splitting graph is chain-free. In other words, chain-freeness is the most precise criterion of decidability of string formulae based on relation-concatenation skeletons (that is, a criterion independent of the particular values of relational, membership, and length constraints).

5 Weakly Chaining to Chain-Free

In the following, we show that, given a weakly chaining formula, we can transform it to an equisatisfiable chain-free formula.

Theorem 1. *A weakly chaining formula can be transformed to an equisatisfiable chain-free formula.*

The rest of this section is devoted to the proof of Theorem 1 (which also provides an algorithm how to transform any weakly chaining formula into an equisatisfiable chain-free formula). In the following, we assume w.l.o.g. that the given weakly-chaining formula Ψ is conjunctive. The proof is done by induction on the number $\mathbf{B}$ of relational constraints that are labelling the set of benign chains in the splitting graph of Ψ.

Base Case ($\mathbf{B} = \mathbf{0}$*).* Since there is no benign chain in G_Ψ, Theorem 1 holds.

Induction Case ($\mathbf{B} > \mathbf{0}$*).* In the following, we will show how to remove one benign chain (and its set of labelling relational constraints) in the case where the splitting graph of Ψ does not contain nested chains. If nested chains are present, then the proof follows the same main ideas, but the reasoning is generalised from one benign chain to strongly connected components. Let $\rho = (p_0, p_1), (p_1, p_2), \ldots, (p_n, p_0)$ be a benign

chain in the splitting graph G_Ψ. For every $i \in \{0,\ldots,n\}$, let $R_i(x_i,t_i)$ be the length preserving relational constraint to which the position p_i belongs. We assume w.l.o.g.[2] that all the positions $p_0, p_1, \ldots, p_n$ are left positions. Since ρ is a benign chain, we have that the variable x_i is appearing in the string term $t_{(i+n) mod (n+1)}$ for all $i \in \{0,\ldots,n\}$. Furthermore, we can use the fact that the relational constraints are length preserving to deduce that the variables $x_0, x_1, \ldots, x_n$ have the same length. This implies also that, for every $i \in \{0,1,\ldots,n\}$, the string term t'_i that is constructed by removing from t_i one occurrence of $x_{(i+1) mod (n+1)}$ is equivalent to the empty word. Therefore, the relational constraint $R_i(x_i,t_i)$ can be rewritten as $R_i(x_i, x_{(i+1) mod (n+1)})$ for all $i \in \{0,1,\ldots,n\}$.

Let $x_{i_1}, x_{i_2}, \ldots, x_{i_k}$ be the maximal subsequence of pairwise distinct variables in $x_0, x_1, \ldots, x_n$. Let `index` be a mapping that associates to each index $\ell \in \{0,\ldots,n\}$ the index $j \in \{1,\ldots,k\}$ such that $x_\ell = x_{i_j}$. We can transform the transducer R_i, with $i \in \{0,\ldots,n\}$, to a length preserving k-tape automaton A_i such that a word $(w_1, w_2, \ldots, w_k)$ is accepted by A_i if and only if $(w_{\mathtt{index}(i)}, w_{\mathtt{index}((i+1) mod (n+1))})$ is accepted by R_i. Let then A be the k-tape automaton resulting from the intersection of $A_0, \ldots, A_n$. Observe that A is also length-preserving. Furthermore, we have that $(w_1, w_2, \ldots, w_k)$ is accepted by A if and only if $(w_{\mathtt{index}(i)}, w_{\mathtt{index}((i+1) mod (n+1))})$ is accepted by R_i for all $i \in \{0,\ldots,n\}$ (i.e., the automaton A characterizes all possible solutions of $\bigwedge_{i=0}^{n} R_i(x_i, x_{(i+1) mod (n+1)}))$. Ideally, we would like to replace the $\bigwedge_{i=0}^{n} R_i(x_i, x_{(i+n) mod (n+1)})$ by $A(x_{i_1}, x_{i_2}, \ldots, x_{i_k})$, however, our syntax forbids such k-ary relation. To overcome this problem, we first extend our alphabet Σ by all the letters in Σ^k and then we replace $\bigwedge_{i=0}^{n} R_i(x_i, x_{(i+1) mod (n+1)})$ by $\varphi := A(x) \wedge \bigwedge_{j=1}^{k} \pi_j(x_{i_j}, x)$ where x is a fresh variable and for every $j \in \{1,\ldots,k\}$, π_j is the length preserving transducer that accepts all pairs of the form $(w_j, (w_1, w_2, \ldots, w_k))$. Finally, let Ψ' be the formula obtained from Ψ by replacing the subformula $\bigwedge_{i=0}^{n} R_i(x_i,t_i)$ in Ψ by $\varphi \wedge |t'_i| = 0$ (remember that the string term t'_i is t_i from which we have removed one occurrence of the variable $x_{(i+1) mod (n+1)}$). It is easy to see that Ψ' is satisfiable iff Ψ is also satisfiable. Furthermore, the number of relational constraints that are labelling the set of benign chains in the splitting graph of Ψ' is strictly less than $\mathbf{B}$ (since π_j can not be used to label any benign chain in Ψ').

6 Chain-Free to Concatenation Free

In the following, we show that we can reduce the satisfiability problem for a chain free formula to the satisfiability problem of a concatenation free formula. To that aim, we describe an algorithm that eliminates concatenation from relational constraints by iterating simple splitting steps. When it terminates, it returns a formula over constraints that are concatenation free. The algorithm can be applied if the string constraints in the formula *allow splitting*, and it is guaranteed to terminate if the formula is *chain-free*. We will explain these two notions below together with the description of the algorithm.

[2] This is possible since if there is benign chain that uses only right positions then there exists also a benign chain that uses only left positions.

Splitting. The *split* of a relational constraint $\varphi ::= R(x \circ t, y \circ t')$ with $t, t' \neq \varepsilon$ is the formula $\Phi_L \vee \Phi_R$ where

$$\Phi_L ::= \bigvee_{i=1}^{n} R_i(x_1, y) \wedge R'_i(x_2 \circ t, t')\ [x/x_1 \circ x_2]$$
$$\Phi_R ::= \bigvee_{j=1}^{m} R_j(x, y_1) \wedge R'_j(t, y_2 \circ t')\ [y/y_1 \circ y_2]$$

$m, n \in \mathbb{N}$, x_1, x_2, y_1, y_2 are fresh variables, and $\eta \models \varphi$ if and only if there is an assignment $\eta' : \{x_1, x_2, y_1, y_2\} \to \Sigma^*$ such that $\eta \cup \eta' \models (\Phi_L \wedge x = x_1 \circ x_2) \vee (\Phi_R \wedge y = y_1 \circ y_2)$. The formula Φ_L is called the *left split* and Φ_R is called the *right split* of φ. In case $t' = \varepsilon$, the split is defined in the same way but with Φ_L left out, and if $t = \varepsilon$, then Φ_R is left out. If both t and t' equal ε, then φ is concatenation free and does not have a split. A simple example is the equation $xy = zz$ with the split $(x_1 = z \wedge x_2 y = z) \vee (x = z_1 \wedge y = z_2 z_1 z_2)$. A class of relational constraints $\mathcal{C}$ *allows splitting* if for every constraint in $\mathcal{C}$ that is not concatenation free, it is possible to compute a split that belongs to $\mathcal{C}$. Equalities as well as transducer constraints allow splitting. A left split of an equality $x \circ t = y \circ t'$ is $x_1 = y \wedge x_2 \circ t = t'$. A left split of a transducer constraint $\mathcal{T}(x \circ t, y \circ t')$ is the formula

$$\bigvee_{q \in Q} \mathcal{T}_q(x_1, y) \wedge {}_q\mathcal{T}(x_2 \circ t, t')$$

where Q is the set of states of $\mathcal{T}$, and ${}_q\mathcal{T}$ and $\mathcal{T}_q$ are the $\mathcal{T}$ with the original set of initial and final states, respectively, replaced by $\{q\}$ (this is the automata splitting technique of [4] extended to transducers in [20]). The right splits are analogous.

Splitting Algorithm. A *splitting algorithm* for eliminating concatenation iterates *splitting steps* on a formula in DNF. A *splitting step* can be applied to one of the clauses if it can be written in the form $\Upsilon ::= \varphi \wedge \Psi$ where $\varphi ::= R(x \circ t, y \circ t')$. It then replaces the clause by a DNF of the disjunction

$$(\Phi_L \wedge \Psi[x/x_1 \circ x_2] \wedge |x| = |x_1| + |x_2|) \vee (\Phi_R \wedge \Psi[y/y_1 \circ y_2] \wedge |y| = |y_1| + |y_2|)$$

where Φ_L and Φ_R are the left and the right split, respectively, of φ. The left or the right disjunct is omitted if $t' = \varepsilon$ or $t = \varepsilon$, respectively. The splitting step is not applied when both t and t' equal ε, i.e. φ is concatenation free. In order to ensure termination, the algorithm applies splitting steps under the following regimen consisting of two phases.

In Phase 1, the algorithm maintains each clause Υ of a DNF of the string formula annotated with a *reminder*, a sub-graph H_Υ of its splitting graph G_Υ. The reminders restrict the choice of splitting steps so that a splitting step can be applied to a clause $\Upsilon = \varphi \wedge \Psi$ only if φ is a *root constraint* in H_Υ, meaning that all positions at one of the sides of φ are root nodes of H_Υ. The reminder graphs are assigned to clauses as follows. The algorithm is initialised with $H_\Upsilon ::= G_\Upsilon$ for each clause Υ. After taking a splitting step, the reminder graph of each new clause Υ' is a sub-graph $H_{\Upsilon'}$ of its splitting graph $G_{\Upsilon'}$. Particularly, $H_{\Upsilon'}$ contains only those constraints of Υ' (their positions that is) that are non-concatenation-free successors of the constraints of Υ that appear in H_Υ. The newly created concatenation free constraints do not propagate to $H_{\Upsilon'}$. Here, by saying that a constraint φ' of Υ' is a *successor* of a constraint φ of Υ means that either $\varphi' = \varphi[x/x_1 \circ x_2]$ or $\varphi' = \varphi[y/y_1 \circ y_2]$, or that they are the constraints explicitly mentioned in the definition of left or right split. Phase 1 terminates when the reminder graphs of all clauses are empty.

Phase 2 then performs splitting steps in any order until all constraints are concatenation free.

Theorem 2. *When run on a chain-free formula, the splitting algorithm terminates with an equisatisfiable chain and concatenation-free formula.*

Hereafter, we provide a brief overview of the proof of Theorem 2. The main difficulty with proving termination of splitting is the substitution of variables involved in the left and right split. The left split makes a step towards concatenation freeness by removing one concatenation operator $\circ$ from the clause, since the terms $x \circ t$ and $y \circ t'$ are replaced by x_1, y, t', and $x_2 \circ t$. However, the substitution of x by $x_1 \circ x_2$ in the reminder of the clause introduces as many new concatenations as there are occurrences of x other than the one explicit in the definition of the left split (and similarly for the right split). Therefore, to guarantee termination of splitting, we must limit the effect of substitution by enforcing chain-freeness.

Why chain-freeness is the right property here may be intuitively explained as follows. The splitting graph of a clause is in fact a map of how chains of substitutions may increase the number of concatenations in the clause. Consider an edge in the splitting graph from a position p to a position p'. By definition, there is an intermediate position p'' opposite p and carrying the same variable as p'. This means that when splitting decreases the number of concatenations on the side of p by one (the label of p may be y referred to in the left split), the substitution of the label of p'' (this would be x in the left split) would cause that the position p' also labeled by x is replaced by the concatenation $x_1 \circ x_2$. Moreover, since the length of the side of p' is now larger, it is possible to perform more splitting steps that follow edges starting at the side of p' and increase numbers of $\circ$ at positions reachable from p' and consequently also further along the path in the splitting graph starting at (p, p'). Hence the intuitive meaning of the edge is that decreasing the number of $\circ$ at the side of p might increase the number of $\circ$ at the side of p'. Chain-freeness now guarantees that it can happen only finitely many times that decreasing the number of $\circ$ at the side of a position p can through a sequence of splitting steps lead to increasing this number.

7 Satisfiability of Chain and Concatenation-Free Formula

In this section, we explain a decision procedure of a chain and concatenation-free formula. The algorithm is essentially a combination of two standard techniques. First, concatenation and chain-free conjunction over relational constraints is a formula in the "acyclic fragment" of [6] that can be turned into a single equivalent transducer constraint (an approach used also in e.g. [14]). Second, consistency of the resulting transducer with the input length constraints may be checked via computation of the Parikh image of the transducer.

We will now describe the two steps in a more detail. For simplicity, we will assume only transducer and length constraints. This is without loss of generality because the other types of constraints can be encoded to transducers.

Transducer Constraints. A conjunction of transducer constraints may be decided through computing an equisatisfiable multi-tape transducer constraint and checking

emptiness of its language. The transducer constraint is computed by synchronizing pairs of constraints in the conjunction. That is, synchronization of two transducer constraints $\mathcal{T}_1(x_1,\dots,x_n)$ and $\mathcal{T}_2(y_1,\dots,y_m)$ is possible if they share at most one variable (essentially the standard automata product construction where the two transducers synchronise on the common variable). The result of their synchronization is then a constraint $\mathcal{T}_1 \wedge_{(i,j)} \mathcal{T}_2(x_1,\dots,x_n,y_1\dots y_{j-1},y_{j+1},\dots,y_m)$ where y_j is the common variable equal to x_i for some $1 \leq i \leq n$ or a constraint $\mathcal{T}_1 \wedge \mathcal{T}_2(x_1,\dots,x_n,y_1,\dots,y_m)$ if there is no common variable. The $\mathcal{T}_1 \wedge \mathcal{T}_2$ is a lose version of $\wedge_{(i,j)}$ that does not synchronise the two transition relations (see e.g. [14,20] for details on implementation of a similar construction). Since the original constraint is chain and concatenation-free, two constraints may share at most one variable. This property is an invariant under synchronization steps, hence they may be preformed in any order until only single constraint remains. Termination of this procedure is immediate because every step decreases the number of constraints.

Length Constraints. A formula of the form $\Psi_r \wedge \Psi_l$ where Ψ_r is a conjunction of relational constraints and Ψ_l is a conjunction of length constraints may be decided through replacing Ψ_r by a Presburger formula Ψ'_r over length constraints that captures the length constraints implied by Ψ_r. That is, an assignment $\nu : \{|x| \mid x \in \mathbb{X}\} \to \mathbb{N}$ is a solution of Ψ'_r if and only if there is a solution η of Ψ_r such that $|\eta(x)| = \nu(|x|)$ for all $x \in \mathbb{X}$. The conjunction $\Psi'_r \wedge \Psi_l$ is then an existential Presburger formula equisatisfiable to the original conjunction, solvable by an of-the-shelf SMT solver.

Construction of Ψ'_r is based on computation of the Parikh image of the synchronised constraint $\mathcal{T}(x_1,\dots,x_n)$ equivalent to Ψ_r. Since $\mathcal{T}$ is a standard finite automaton over the alphabet of n-tuples Σ_ε^n, its Parikh image can be computed in the form of a semi-linear set represented as an existential Presburger formula Ψ_{Parikh} by a standard automata construction (see e.g. [32]). The formula captures the relationship between the numbers of occurrences of letters of Σ_ε^n in words of $\mathcal{L}(\mathcal{T})$. Particularly, the numbers of letter occurrences are represented by the *Parikh variables* $\mathbb{P} = \{\#\alpha \mid \alpha \in \Sigma_\varepsilon^n\}$ and it holds that The formula$\nu \models \Psi_{\text{Parikh}}$ iff there is a word $w \in \mathcal{L}(\mathcal{T})$ such that for all $\alpha \in \Sigma_\varepsilon^n$, α appears $\nu(\#\alpha)$ times in w.

The formula Ψ'_r is then extracted from Ψ_{Parikh} as follows. Let $\mathbb{A} = \{\#a_i \mid a \in \Sigma, 1 \leq i \leq n\}$ be a set of *auxiliary variables* expressing how many times the letter $a \in \Sigma$ appears on the ith position of a symbol from Σ_ε^n in a word from $\mathcal{L}(\mathcal{T})$. Let $\alpha[i]$ denotes the ith component of the tuple $\alpha \in \Sigma_\varepsilon^n$. We construct the formula Φ that uses variables $\mathbb{A}$ to describe the relation between values of $|x_1|,\dots,|x_n|$ and variables of $\mathbb{P}$:

$$\Phi := \bigwedge_{i=1}^{n} \Big(|x_i| = \sum_{a\in\Sigma} \#a_i \wedge \bigwedge_{a\in\Sigma} \Big(\#a_i = \sum_{\alpha\in\Sigma_\varepsilon^n \text{ s.t. } \alpha[i]=a} \#\alpha \Big) \Big)$$

We the obtain Ψ'_r by eliminating the quantifiers from $\exists\mathbb{P}\exists\mathbb{A} : \Phi \wedge \Psi_{\text{Parikh}}$.

8 Experimental Results

We have implemented our decision procedure in SLOTH [14] and then used it in the over-approximation module of the string solver TRAU+, which is an extension of TRAU[3]. TRAU+ is as an open source string solver and used Z3 [11] as the SMT solver to handle generated arithmetic constraints. TRAU+ is based on a Counter-Example

Table 1. Results of running solvers over Chain-Free, two sets of the SLOG, and four sets of PyEx suite.

		Ostrich	Z3-str3	CVC4	TRAU	TRAU+
Chain-Free (26)	sat	0	-	-	-	**5**
	unsat	0	-	-	-	**14**
	timeout	6	-	-	-	7
	error/unknown	20	-	-	-	0
ReplaceAll (120)	sat	**106**	-	-	26	105
	unsat	**14**	-	-	4	**14**
	timeout	0	-	-	0	1
	error/unknown	0	-	-	90	0
Replace (3392)	sat	1250	298	1278	1174	**1287**
	unsat	2022	2075	2079	2080	**2081**
	timeout	3	903	9	24	23
	error/unknown	117	116	26	114	1
PyEx-td (5569)	sat	36	839	4178	4244	**4245**
	unsat	299	1477	1281	1287	**1287**
	timeout	0	3027	105	35	35
	error/unknown	5234	226	5	3	2
PyEx-z3 (8414)	sat	35	1211	5617	6680	**6681**
	unsat	466	**1870**	1346	1357	1357
	timeout	0	4760	1449	374	374
	error/unknown	7913	573	2	3	2
PyEx-zz (11438)	sat	38	2840	**9817**	8966	8967
	unsat	141	1974	**1202**	1192	1193
	timeout	0	5988	416	1277	1276
	error/unknown	11259	636	3	3	2
Total (28959)	solved	4407	12730	26798	27010	**27236**
	unsolved	24552	16229	2161	1949	**1723**

Guided Abstraction Refinement (CEGAR) framework which contains both an under- and an over-approximation module. These two modules interact together in order to automatically make these approximations more precise. The extension of SLOTH in the over-approximation module of TRAU+ takes as an input a constraint and checks if it belongs to the weakly-chaining fragment. If it is the case, then we use our decision procedure outlined above. Otherwise, we start by choosing a minimal set of occurrences of variables x that needs to be replaced by fresh ones such that the resulting constraint falls in our decidable fragment.

We compare TRAU+ performance against the performance of four other state-of-the-art string solvers, namely Ostrich [10], Z3-str3 [7], CVC4 1.6 [18,19], and TRAU [1]. For our comparison with Z3-str3, we use the version that is part of Z3 4.8.4. The goal of our experiments is twofold:

– TRAU+ handles transducer constraints in an efficient manner TRAU+ can handle more cases than TRAUsince the new over-approximation of TRAU+ supports more and new transducer constraints that the one of TRAU.
– TRAU+ performs either better or as well as existing tools on transducer-less benchmarks.

We carry experiments on suites that draw from the real world applications with diverse characteristics. The first suite is our new suite Chain-Free. Chain-Free is obtained from variations of various PHP code, including the introductory example. The second suite is SLOG [36] that is derived from the security analysis of real web applications. The suite was generated by Ostrich group. The last suite is PyEx [27] that is derived from PyEx - a symbolic executor designed to assist Python developers to achieve high coverage testing. The suite was generated by CVC4 group on 4 popular Python packages: `httplib2`, `pip`, `pymongo`, and `requests`. The summary of experimenting Chain-Free, SLOG, and PyEx is given in Table 1. All experiments were performed on an Intel Core i7 2.7 Ghz with 16 GB of RAM. The time limit is 30s for each test which is widely used in the evaluation of other string solvers. Additionally, we use 2700 s for Chain-Free suite - much larger than usual as its constraints are difficult. Rows with heading "sat"("unsat") indicate the number of times the solver returned satisfiable (unsatisfiable). Rows with heading "timeout" indicate the number of times the solver exceeded the time limit. Rows with heading "error/unknown" indicate the number of times the solver either crashed or returned unknown.

Chain-Free suite consists of 26 challenging chain-free tests, 6 of them being also straight-line. The tests contain Concatenation, ReplaceAll, and general transducers constraints encoding various JavaScript and PHP functions such as htmlescape, escapeString. Since Z3-str3, CVC4, and TRAU do not support the language of general transducers, we skip performing experiments on those tools in the suite. Ostrich returns 6 times "timeout" for straight-line tests, and times 20 "unknown" for the rest. TRAU+ handles well most cases, and gets "timeout" for only 7 tests.

SLOG suite consists of 3512 tests which contain transducer constraints such as Replace and ReplaceAll. Since Z3-str3 and CVC4 do not support the ReplaceAll function, we skip doing experiments on those tools in the ReplaceAll set. In both sets, the result shows that TRAU+ clearly improved TRAU. In particular, TRAU+ can handle most cases where TRAU returns either "unknown" and "timeout". TRAU+ has also better performance than other solvers.

PyEx suite consists of 25421 tests which contain diverse string constraints such as IndexOf, CharAt, SubString, Concatenation. TRAU and CVC4 have similar performance on the suite. While TRAU is better on PyEx-dt and PyEx-z3 sets (3 less error/unknown results, roughly 1000 less timeouts), CVC4 is better on PyEx-zz set (about 800 less timeouts). CVC4 and TRAU clearly have an edge over Z3-str3 in all aspects. Comparing with Ostrich on this benchmark is problematic because it mostly fails due to unsupported syntactic features. TRAU+ is better than TRAU on all three benchmark sets. This shows that our proposed procedure is efficient in solving not only transducer examples, but also in transducer-less examples.

To summarise our experimental results, we can see that:

- TRAU+ handles more transducer examples in an efficient manner. This is illustrated by the Chain-Free and Slog suites. The experiment results on these benchmarks show that TRAU+ outperforms TRAU. Many tests on which TRAU returns "unknown" are now successfully handled by TRAU+ .
- TRAU+ performs as well as existing tools on transducer-less benchmarks and in fact sometimes TRAU+ outperforms them. This is illustrated by the PyEx suite. In fact, this benchmark is handled very well by TRAU, but nevertheless, as evident from the table, our tool is doing better than TRAU. In fact, As TRAU+ out-performs TRAU in some examples in the PyEx suite. In those examples, TRAU returned "unknown' while TRAU+ returned "unsat". This means that the new over-approximation not only improves TRAU in transducer benchmarks, but it also improves TRAU in transducer-less examples. Furthermore, observe that the PyEx suite has only around 4000 "unsat" cases out of 25k cases.

9 Related Work

Already in 1946, Quine [26] showed that the first order theory of string equations is undecidable. An important line of work has been to identify subclasses for which decidability can be achieved. The pioneering work by Makanin [21] proposed a decision procedure for quantifier-free word equations, i.e., Boolean combinations of equalities and disequalities, where the variables may denote words of arbitrary lengths. The decidability and complexity of different subclasses have been considered by several works, e.g. [12,13,22,24,25,28,31]. Generalizations of the work of Makanin by adding new types of constraints have been difficult to achieve. For instance, the satisfiability of word equations combined with length constraints of the form $|x| = |y|$ is open [8]. Recently, regular and especially relational transducers constraints were identified as a strongly desirable feature of string languages especially in the context software analysis with an emphasis on security. Adding these to the mix leads immediately to undecidability [23] and hence numerous decidable fragments were proposed [4,6,9,10,20]. From these, the straight line fragment of [20] is the most general decidable combination of concatenation and transducers. It is however incomparable to the acyclic fragment of [4] (which does not have transducers but could be extended with them in a straightforward manner). Some works add also other syntactic features, such as [9,10], but the limit of decidable combinations of the core string features—transducers/regular constraints, length constraints, and concatenation stays at [20] and [4]. The weakly chaining decidable fragment present in this paper significantly generalises both these fragments in a practically relevant direction.

The strong practical motivation in string solving led to a rise of a number of SMT solvers that do not always provide completeness guarantees but concentrate on solving practical problem instances, through applying a variety of calculi and algorithms. A number of tools handle string constraints by means of *length-based under-approximation* and translation to bit-vectors [17,29,30], assuming a fixed upper bound on the length of the possible solutions. Our method on the other hand allows to analyse

constraints without a length limit and with completeness guarantees. More recently, also *DPLL(T)-based* string solvers lift the restriction of strings of bounded length; this generation of solvers includes Z3-str3 [7], Z3-str2 [38], CVC4 [18], S3P [33,34], Norn [5], Trau [3], Sloth [14], and Ostrich [10]. DPLL(T)-based solvers handle a variety of string constraints, including word equations, regular expression membership, length constraints, and (more rarely) regular/rational relations; the solvers are not complete for the full combination of those constraints though, and often only decide a (more or less well-defined) fragment of the individual constraints. Equality constraints are normally handled by means of splitting into simpler sub-cases, in combination with powerful techniques for Boolean reasoning to curb the resulting exponential search space. Our implementation is combining strong completeness guarantees of [14] extended to handle the fragment proposed in this paper with an efficient approximation techniques of [3] and its performance on existing benchmarks compares favourably with the most efficient of the above tools.

A further direction is *automata-based* solvers for analyzing string-manipulated programs. Stranger [37] soundly over-approximates string constraints using regular languages, and outperforms DPLL(T)-based solvers when checking single execution traces, according to some evaluations [16]. It has recently also been observed [14,36] that automata-based algorithms can be combined with model checking algorithms, in particular IC3/PDR, for more efficient checking of the emptiness for automata. However, many kinds of constraints, including length constraints and word equations, cannot be handled by automata-based solvers in a complete manner.

References

1. Abdulla, P.A., et al.: Trau String Solver. https://github.com/diepbp/FAT
2. Abdulla, P.A., et al.: Flatten and conquer: a framework for efficient analysis of string constraints. In: PLDI. ACM (2017)
3. Abdulla, P.A., et al.: Trau: SMT solver for string constraints. In: FMCAD. IEEE (2018)
4. Abdulla, P.A., et al.: String constraints for verification. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 150–166. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_10
5. Abdulla, P.A., et al.: Norn: an SMT solver for string constraints. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 462–469. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_29
6. Barceló, P., Figueira, D., Libkin, L.: Graph logics with rational relations. Logical Methods Comput. Sci. **9**(3) (2013). https://doi.org/10.2168/LMCS-9(3:1)2013
7. Berzish, M., Zheng, Y., Ganesh, V.: Z3str3: a string solver with theory-aware branching. CoRR abs/1704.07935 (2017)
8. Büchi, J.R., Senger, S.: Definability in the existential theory of concatenation and undecidable extensions of this theory. Z. Math. Logik Grundlagen Math. **34**(4) (1988)
9. Chen, T., Chen, Y., Hague, M., Lin, A.W., Wu, Z.: What is decidable about string constraints with the replace all function. Proc. ACM Program. Lang. **2**(POPL) (2018). https://doi.org/10.1145/3158091
10. Chen, T., Hague, M., Lin, A.W., Rümmer, P., Wu, Z.: Decision procedures for path feasibility of string-manipulating programs with complex operations. Proc. ACM Program. Lang. **3**(POPL) (2019). https://doi.org/10.1145/3290362

11. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
12. Ganesh, V., Berzish, M.: Undecidability of a theory of strings, linear arithmetic over length, and string-number conversion. CoRR abs/1605.09442 (2016)
13. Ganesh, V., Minnes, M., Solar-Lezama, A., Rinard, M.: Word equations with length constraints: what's decidable? In: Biere, A., Nahir, A., Vos, T. (eds.) HVC 2012. LNCS, vol. 7857, pp. 209–226. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39611-3_21
14. Holík, L., Janku, P., Lin, A.W., Rümmer, P., Vojnar, T.: String constraints with concatenation and transducers solved efficiently. PACMPL **2**(POPL) (2018). https://doi.org/10.1145/3158092
15. Hu, Q., D'Antoni, L.: Automatic program inversion using symbolic transducers. In: SIGPLAN Notices, vol. 52, no. 6, June 2017
16. Kausler, S., Sherman, E.: Evaluation of string constraint solvers in the context of symbolic execution. In: ASE 2014. ACM (2014)
17. Kiezun, A., Ganesh, V., Guo, P.J., Hooimeijer, P., Ernst, M.D.: HAMPI: a solver for string constraints. In: ISTA 2009. ACM (2009)
18. Liang, T., Reynolds, A., Tinelli, C., Barrett, C., Deters, M.: A DPLL(T) theory solver for a theory of strings and regular expressions. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 646–662. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_43
19. Liang, T., Reynolds, A., Tinelli, C., Barrett, C., Deters, M.: CVC4 (2016). http://cvc4.cs.nyu.edu/papers/CAV2014-strings/
20. Lin, A.W., Barceló, P.: String solving with word equations and transducers: towards a logic for analysing mutation XSS. In: POPL 2016. ACM (2016)
21. Makanin, G.: The problem of solvability of equations in a free semigroup. Math. USSR-Sbornik **32**(2) (1977)
22. Matiyasevich, Y.: Computation paradigms in light of Hilbert's tenth problem. In: Cooper, S.B., Löwe, B., Sorbi, A. (eds.) New Computational Paradigms, pp. 59–85. Springer, New York (2008). https://doi.org/10.1007/978-0-387-68546-5_4
23. Morvan, C.: On rational graphs. In: Tiuryn, J. (ed.) FoSSaCS 2000. LNCS, vol. 1784, pp. 252–266. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-46432-8_17
24. Plandowski, W.: Satisfiability of word equations with constants is in PSPACE. J. ACM **51**(3) (2004)
25. Plandowski, W.: An efficient algorithm for solving word equations. In: STOC 2006. ACM (2006)
26. Quine, W.V.: Concatenation as a basis for arithmetic. J. Symb. Log. **11**(4) (1946)
27. Reynolds, A., Woo, M., Barrett, C., Brumley, D., Liang, T., Tinelli, C.: Scaling up DPLL(T) string solvers using context-dependent simplification. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10427, pp. 453–474. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63390-9_24
28. Robson, J.M., Diekert, V.: On quadratic word equations. In: Meinel, C., Tison, S. (eds.) STACS 1999. LNCS, vol. 1563, pp. 217–226. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-49116-3_20
29. Saxena, P., Akhawe, D., Hanna, S., Mao, F., McCamant, S., Song, D.: A symbolic execution framework for JavaScript. In: IEEE Symposium on Security and Privacy. IEEE (2010)
30. Saxena, P., Hanna, S., Poosankam, P., Song, D.: FLAX: systematic discovery of client-side validation vulnerabilities in rich web applications. In: NDSS. The Internet Society (2010)

31. Schulz, K.U.: Makanin's algorithm for word equations-two improvements and a generalization. In: Schulz, K.U. (ed.) IWWERT 1990. LNCS, vol. 572, pp. 85–150. Springer, Heidelberg (1992). https://doi.org/10.1007/3-540-55124-7_4
32. Seidl, H., Schwentick, T., Muscholl, A., Habermehl, P.: Counting in trees for free. In: Díaz, J., Karhumäki, J., Lepistö, A., Sannella, D. (eds.) ICALP 2004. LNCS, vol. 3142, pp. 1136–1149. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27836-8_94
33. Trinh, M.T., Chu, D.H., Jaffar, J.: S3: a symbolic string solver for vulnerability detection in web applications. In: CCS 2014. ACM (2014)
34. Trinh, M.-T., Chu, D.-H., Jaffar, J.: Progressive reasoning over recursively-defined strings. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9779, pp. 218–240. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41528-4_12
35. TwistIt.tech: PHP tutorials (2019). https://www.makephpsites.com/php-tutorials/user-management-tools/changing-passwords.php. Accessed 29 Apr 2019
36. Wang, H.-E., Tsai, T.-L., Lin, C.-H., Yu, F., Jiang, J.-H.R.: String analysis via automata manipulation with logic circuit representation. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9779, pp. 241–260. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41528-4_13
37. Yu, F., Alkhalaf, M., Bultan, T.: Stranger: an automata-based string analysis tool for PHP. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 154–157. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12002-2_13
38. Zheng, Y., Zhang, X., Ganesh, V.: Z3-str: a Z3-based string solver for web application analysis. In: ESEC/FSE 2013. ACM (2013)

Efficient Handling of String-Number Conversion

Parosh Aziz Abdulla
Uppsala University
Uppsala, Sweden
parosh@it.uu.se

Mohamed Faouzi Atig
Uppsala University
Uppsala, Sweden
mohamed_faouzi.atig@it.uu.se

Yu-Fang Chen
Academia Sinica
Taipei, Taiwan
yfc@iis.sinica.edu.tw

Bui Phi Diep
Uppsala University
Uppsala, Sweden
bui.phi-diep@it.uu.se

Julian Dolby
IBM Research
NY, USA
dolby@us.ibm.com

Petr Janků
Brno University of Technology
Brno, Czechia
ijanku@fit.vutbr.cz

Hsin-Hung Lin
Academia Sinica
Taipei, Taiwan
hlin@iis.sinica.edu.tw

Lukáš Holík
Brno University of Technology
Brno, Czechia
holik@fit.vutbr.cz

Wei-Cheng Wu
University of Southern California
CA, USA
wwu@isi.edu

Abstract

String-number conversion is an important class of constraints needed for the symbolic execution of string-manipulating programs. In particular solving string constraints with string-number conversion is necessary for the analysis of scripting languages such as JavaScript and Python, where string-number conversion is a part of the definition of the core semantics of these languages. However, solving this type of constraint is very challenging for the state-of-the-art solvers. We propose in this paper an approach that can efficiently support both string-number conversion and other common types of string constraints. Experimental results show that it significantly outperforms other state-of-the-art tools on benchmarks that involves string-number conversion.

CCS Concepts: • **Security and privacy** → **Logic and verification**; • **Software and its engineering** → *Formal methods.*

Keywords: String Solver, Formal Verification, Automata

ACM Reference Format:
Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Bui Phi Diep, Julian Dolby, Petr Janků, Hsin-Hung Lin, Lukáš Holík, and Wei-Cheng Wu. 2020. Efficient Handling of String-Number Conversion. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '20), June 15–20, 2020, London, UK*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3385412.3386034

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PLDI '20, June 15–20, 2020, London, UK
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7613-6/20/06…$15.00
https://doi.org/10.1145/3385412.3386034

1 Introduction

Symbolic execution is a very popular technique that allows programmers to check the feasibility of a path in a program, i.e., determining the value of the inputs under which the given path can be executed. The path feasibility problem is usually solved by a reduction to the satisfiability of a formula. More precisely, program statements in the path are translated to equivalent constraints in static single assignment (SSA) form and then solved by *Satisfiability Modulo Theory (SMT)* solvers. The types of constraints needed depend on the types of program expressions to be analyzed. Therefore, SMT solvers need to support different combinations of theories so that they can handle a wide range of types.

Among all data types, the *string data type* is omnipresent in modern programming languages. Various security vulnerabilities such as injection and cross-site scripting attack are caused by malicious string values. Therefore, string constraint solving has received considerable attention in the constraint solving community. Operations such as *equality constraints* (e.g. $x.y = y.x$), *regular constraints* (e.g., $x \in (a.b)^*$), and *integer constraints* (e.g., $|x| - |y| > 3$), are widely supported by most state-of-the-art string constraint solvers such as, CVC4 [8], OSTRICH [13], Sloth [21], Trau+ [1, 2, 5], Z3 [15] and Z3Str3 [9].

An important class of string operations is the string-number conversions. While string length operations are sometimes well supported, converting a string x to an integer n (e.g., using the operation $n = \text{toNum}(x)$) or turning an integer value n into its string form x (e.g., using the operation $x = \text{toStr}(n)$) suffer from limited support (in terms of the scale of formulae they can handle) by the state-of-the-art string constraint solvers.

In fact, a code that receives string input tends to need to convert at least some of that input into numbers. For example, the program fragment below is a variant of the Luhn test algorithm that is often used in credit card or ID validation.

```
function checkLuhn(value) {
   var sum = 0;
   for (var i = value.length - 1; i >= 0; i-=2) {
       var d = parseInt(value.charAt(i));
       sum += d;
   }
   for (var i = value.length - 2; i >= 0; i-=2) {
       var d = parseInt(value.charAt(i));
       if ((d *= 2) > 9) d -= 9;
       sum += d;
   }
   var last= sum.toString().charAt(sum.length-1);
   return last == '0';
}
```

The input `value` of the Luhn test algorithm is a sequence of digits. The algorithm processes the digits in the reversed order. The value of every odd digit (e.g., 1st, 3rd, etc.) is added to `sum` directly. For the value of every even digit, the algorithm (1) doubles its value, (2) subtracts its value by 9 if the doubled-value is larger than 9, and (3) adds the final result to `sum`. At the end, the input is validated if the last digit of `sum` is 0 (i.e., `sum mod 10=0`).

To check whether the program path that traverses both loops exactly once and finally passes this test has a valid input, we create the following (string) constraint:

1	$value_0 \in [1,9]^+ \wedge sum_0 = 0 \wedge$
2	$i_0 = \|value_0\| - 1 \wedge$
3	$d_0 = \mathrm{toNum}(\mathrm{charAt}(value_0, i_0)) \wedge$
4	$sum_1 = sum_0 + d_0 \wedge$
5	$i_1 = \|value_0\| - 2 \wedge$
6	$d_1 = \mathrm{toNum}(\mathrm{charAt}(value_0, i_1)) \wedge$
7	$sum_2 = sum_1 + \mathrm{ite}(d_1 * 2 > 9, d_1 * 2 - 9, d_1 * 2) \wedge$
8	$i_2 = 0$
9	$last_0 = \mathrm{charAt}(\mathrm{toStr}(sum_2), \|\mathrm{toStr}(sum_2)\| - 1) \wedge$
10	$last_0 = \text{“0”}$

Here $value_0$ and $last_0$ are string variables and the others are integer variables. The method $\mathrm{charAt}(x, i)$ returns the character at index i in the string x while $n = \mathrm{ite}(b, e, e')$ assigns to n the value of the expression e if b is true and the value of the expression e' otherwise. Line 1 describes the initial condition: value should be a sequence of digits and sum is initially zero. Lines 2-4 and lines 5-7 describe one execution of the first and second loop, respectively. Line 8 describes the condition on i_2 before leaving the loop. Finally, Lines 9-10 describe the condition that the last digit of sum is zero. Observe that to describe such a program path, we need a solver that supports the following types of constraints:

- *Regular* constraints (e.g., $value_0 \in [1,9]^+$, which says $value_0$ is in the regular language $[0,9]^+$),
- *Integer* constraints (e.g., $i_0 = |value_0| - 1$, which says i_0 equals the length of $value_0$ minus one),
- *Equality* constraints (often $y = \mathrm{charAt}(x, i)$ is encoded as $x = x_1.x_2.x_3 \wedge |x_1| = i \wedge |x_2| = 1 \wedge y = x_2$, which uses equality of string terms x and $x_1.x_2.x_3$), and
- *String-number conversion* (e.g., $\mathrm{toStr}(sum_2)$, which is the string value of the number sum_2).

Most of the state-of-the-art string constraint solvers provide limited support to the combination of above constraints. In Table 3 of our evaluation (Section 9), CVC4 fails to solve constraints corresponding to checkLuhn of more than 6 loop iterations in 2 minutes, Z3 can only solve the cases corresponding to 2 to 5, and 9 loop iterations, and Z3Str3 fails to solve any case.

Even, more crucially, in many programming languages, string-number conversion is a part of the definition of their core semantics. JavaScript, which powers most interactive content on the Web and increasingly server-side code with Node.js, is one of such languages. Other scripting languages do too, but we focus on JavaScript due to its prominence. To see how string-integer conversion pervades semantics, consider the following program:

```
for(var i = 0; i < 10; i++) {
    arr[i] = 0;
}
```

A casual glance at the above code reveals no use of strings at all, but the semantics of field access is somewhat unusual in JavaScript. In fact, the arrays in JavaScript are indexed by strings, and numeric indices are converted to strings. This conversion is mandated explicitly by the JavaScript semantics. The 2019 edition of ECMAScript [16] requires that *ToPropertyKey* be called on the element expression (§12.3.2.1), and *ToPropertyKey* calls `ToString` on that value in all but special cases (§7.1.14). Therefore, any faithful symbolic execution of JavaScript must handle such conversions for even basic array operations to work correctly. Consider the following code snippet that manipulates an array `x`, with its value shown on the right:

1	`x = [0,0,0,0,0]`	[0,0,0,0,0]
2	`x[3] = 4`	[0,0,0,4,0]
3	`x[03] = 2`	[0,0,0,2,0]
4	`x["3"] = 5`	[0,0,0,5,0]
5	`x["03"] = 7`	[0,0,0,5,0] and x["03"] = 7
6	`x["03"-1] = 2`	[0,0,2,5,0] and x["03"] = 7

Here `x[3]` in line 2, `x[03]` in line 3, and `x["3"]` in line 4 all denote the same array element of `x["3"]` (due to the implicit conversion of numeric indices to strings in JavaScript), but `x["03"]` denotes a completely different element (which is stored at the index `"03"` of the array). So naïve modeling of array indices with integers will not work – it cannot distinguish the indices `"3"` and `"03"`.

But if array indices are modeled as strings, we must handle arithmetic somehow. Let us look at the case of line 6, we need

to update the value of `x["03"-1]`. The evaluation of the expression `"03"-1` involves an implicit type conversion from the string `"03"` to an integer value 3 due to the `-` (minus) operation. The result of the evaluation of `"03"-1` is the integer 2, which is then converted back to string `"2"` and used as the array index. Hence `x["03"-1]` means the array element of `x["2"]`. Even for a simple example like this, the conversion between string and number is unavoidable. This is a rather basic array operation in JavaScript, and not handling string-number conversion operations will cripple any analysis of non-trivial JavaScript code. Thus, we need stronger solvers that are able to handle string-number conversion operations in order to be able to analyze real code.

Solving string constraint with string-number conversion is a very challenging problem. From the theoretical point of view, this problem is already proven to be undecidable [14]. From practical point of view, our experimental results (in Section 9) show that the current the state-of-the-art string constraint solvers provide little support to string-number conversion.

In this paper, we propose a framework that efficiently handles string constraints with string-number conversion. Since the problem is provably unsolvable, our framework combines over and under-approximation techniques. The over-approximation is for proving UNSAT when possible, while the under-approximation is for proving SAT when possible. Both over- and under-approximation fall in a decidable fragment of string constraints that we can efficiently solve.

For ease of presentation, we use the following toy example

$$\Phi = \{\text{“0”}x = x\text{“0”}, \mathsf{toNum}(x) = \mathsf{toNum}(y), |y| > |x| > 1, 1000 < |y|\}$$

to explain the main ideas behind our decision procedure. To make our terminology explicit: Φ states that “0” concatenated with x is the same as x with “0”, the numeric value of the string x is equivalent to that of y, y is longer than x, y is longer than 1000 characters, and x is longer than 1. Notice that Φ is satisfiable. E.g., it has a model x = “00” and y = “0^{1002}”. Although this toy example is seemingly trivial, all the state-of-the-art string constraint solvers we tried (including Z3, CVC4, and Z3Str3) cannot solve it within 10 minutes.

Our new decision procedure solves the example in few seconds. It proceeds in two steps: The first step consists in over-approximating the set of input constraints into a set that falls in the chain-free fragment [5], which is decidable. Observe that we could over-approximate the input constraint into any decidable fragment, e.g. the acyclic fragment [3] or the straight-line fragment [13]. Our choice of the chain-free fragment [5] is only motivated by the fact that the chain-free fragment is the *largest* known decidable fragment for that class of string constraints. In our example, we over-approximate the formula Φ by converting “0”x = x“0” to two formulae $\{x_1 = \text{“0”}x, x_2 = x\text{“0”}\}$ and replacing the constraint $\mathsf{toNum}(x) = \mathsf{toNum}(y)$ with $n_x = n_y \wedge (n_x = -1 \vee (n_x \neq -1 \wedge x \in [0-9]^*)) \wedge (n_y = -1 \vee (n_y \neq -1 \wedge y \in [0-9]^*))$. Observe

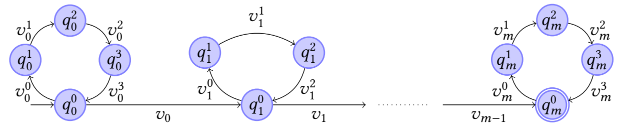


Figure 1. An example of a parametric flat automaton

that if the over-approximation is UNSAT then our decision procedure declares that the original formula is also UNSAT and terminates. Surprisingly, despite its simplicity, our over-approximation procedure works very well in practice as shown by our experimental results (in Section 9). Coming back to the formula Φ, the over-approximation module will return SAT in this case.

The second step of our decision procedure is only enabled if the over-approximation step returns SAT. In this case, our decision procedure uses an under-approximation technique (which is our main contribution) to restrict the search domain of each string variable to strings that obey some predefined and parameterized pattern. We propose to use patterns defined by *parametric flat automata* (PFA). A PFA is a *flat* finite state automaton consisting of a predefined sequence of loops, each of fixed length (see Figure 1). The size of the PFA is parameterized by the length of the sequence of loops and the size of each loop. Adjusting these parameters enlarges or prunes the potential solution space. This approach based on PFA is very flexible yet allows very efficient manipulation. In fact, our procedure restricts the search space for each variable to the set of words accepted by the corresponding given PFA.

Then, we show that given such restriction, one can reduce the string constraint solving problem to a linear formula satisfiability problem in polynomial-time. To gain in efficiency, we label each transition of a PFA with a unique *character* variable (whose domain is the set of natural numbers) instead of having a transition between every two states for each symbol in the alphabet. This is done by associating to each character in our alphabet a unique natural number. This allows us to avoid the *alphabet explosion problem* from which the approach in [1] suffers and it is the key for handling string-number conversion efficiently.

In the following, we explain the construction of the linear formula using Φ as an example. Assume that we project the domains of x and y to the PFA in Figure 2 (a) and (b), respectively. The variables v_0, v_1, v_2, v_3 in the figure are *character* variables. Thus, v_0, v_1, v_2, v_3 are also integer variables.

The linear formula produced after the domain restriction will be over variables v_0, v_1, v_2, v_3, as well as the number of occurrences of each character variable $\#v_0, \#v_1, \#v_2, \#v_3$. Each model of the linear formula encodes a model of the string constraint. For example, x = “00” and y = “0^{1002}” is encoded by the assignment $(v_0, v_1, v_2, v_3, \#v_0, \#v_1, \#v_2, \#v_3) \rightarrow$

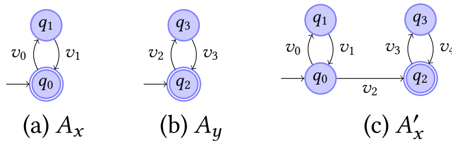


Figure 2. Parametric flat automata of x and y

$(0, 0, 0, 0, 1, 1, 501, 501)$.[1] The assignment says, for example, that x is the *parametric word* obtained by traversing the loop of A_x once (because $\#v_0 = \#v_1 = 1$), which is v_0v_1. Under the assignment $v_0 = 0$ and $v_1 = 0$, we obtain $x =$ "00".

If a model of the produced linear formula is found, then the procedure concludes SAT with an assignment to the string variables. If not, our procedure changes the PFAs to a more expressive one (by adding more states and transitions) and repeat the analysis. We report unknown after failing to prove SAT using a certain number of PFAs.

To demonstrate the usefulness of our approach, we have implemented our decision procedure in an open source solver, called Z3-Trau and evaluated it on a large set of benchmarks obtained from the literature and from symbolic execution of real world programs. The experimental results show that Z3-Trau is among the best tools for solving basic string constraints and significantly outperforms all other tools on benchmarks with string-number conversion constraints. In this benchmark, the total amount of tests cannot be solved by Z3-Trau is only a half to the second best tool.

Summary of the Contributions.

- An *efficient* procedure for checking satisfiability of string constraints with string-number conversion.
- The class of *parametric flat automata* which is the key for efficient handling of string constraints.
- An algorithm that translates the satisfiability problem of string constraints to the satisfiability problem of a linear formula in polynomial-time when the search space restricted by PFAs.
- An open source tool Z3-Trau with experimental results that demonstrate the efficiency of our approach on both existing and real-life benchmarks

Outline. After recalling the definition in Section 3, Section 4 presents a brief overview of our decision procedure. Section 5 introduces the class of parametric flat automata. Section 6 describes how to use PFA to restrict the searching domain of string variables. Section 7 shows how to construct the linear formula for basic string constraints (i.e., regular, equality, and integer constraints). Section 8 presents the construction of the linear formula for string-number conversion operations. Section 9 presents the details of our implementation and our experimental results. Related works are discussed in Section 10. Finally, Section 11 concludes the paper with a discussion of future works.

[1]In these examples, we use the shorthand $(x_1, \ldots, x_k) \to (n_1, \ldots, n_k)$ to denote the function $\{x_1 \to n_1, \ldots, x_k \to n_k\}$.

2 Preliminaries

We use $\mathbb{N}$ and $\mathbb{Z}$ to denote the sets of natural numbers and integers. For a set A, we use $|A|$ to denote its size. For $n, m \in \mathbb{N}$, we write $[n, m]$ for the set of natural numbers $\{k \mid n \leq k \leq m\}$. The function f with the domain restricted to a set D is denoted by f_D, and a set of functions F restricted to a set D is $F_D = \{f_D \mid f \in F\}$. An *alphabet* is a finite set Σ of *characters* and a *word* over Σ is a sequence $w = a_1 \ldots a_n$ of characters from Σ, with ϵ denoting the *empty word*. We use $w_1 \cdot w_2$ to denote the *concatenation* of words w_1 and w_2. Σ^* is the set of all words over Σ, $\Sigma^+ = \Sigma^* \setminus \{\epsilon\}$ and $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$. A *language* over Σ is a subset L of Σ^*. We use $|w|$ to denote the length of w and $|w|_a$ to denote the number of occurrences of the character $a \in \Sigma$ in w.

A *finite automaton* (FA) is a tuple (Q, T, Σ, q_i, q_f), where Q is the set of *states*, $T \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times Q$ is the set of *transitions*, Σ is the alphabet, q_i is the *initial state*, and q_f is the *final state*. A *run* π of A over a word $w = a_1 \cdots a_n$ is a sequence of transitions $(q_0, a_1, q_1), (q_1, a_1, q_2), \ldots, (q_{n-1}, a_n, q_n)$. The run π (resp. the word w) is *accepting* (resp. *accepted*) if $q_0 = q_i$ and $q_n = q_f$. The *language* of A (denoted by $L(A)$) consists of the set of all accepted words.

Through the paper, we will use quantifier-free linear integer arithmetic formulae, and call them *linear formulae* for short. Given a linear formula ϕ over variables V and an *integer interpretation* of V, a function $I : V \longrightarrow \mathbb{Z}$, we denote by $I \models \phi$ that I satisfies ϕ (which is defined in the standard manner), and call I a *model* of ϕ. We use $[\![\phi]\!]$ to denote the set all models of ϕ.

The *Parikh image* of a word $w \in \Sigma^*$ maps each *Parikh variable* $\#a$, where $a \in \Sigma$ is a character, to the number of occurrences of a in w. Formally, given a set S, let $\#S$ denote the set of Parikh variables $\{\#s \mid s \in S\}$. The Parikh image of w is a function $\mathbb{P}(w) : \#\Sigma \longrightarrow \mathbb{N}$ such that $\mathbb{P}(w)(\#a) = |w|_a$ for each $a \in \Sigma$. The Parikh image of a language L is defined as follows $\mathbb{P}(L) = \{\mathbb{P}(w) \mid w \in L\}$. It is well known that the Parikh image of a regular language can be characterized by a linear formula.

Lemma 2.1 ([40]). *Let A be a FA over the alphabet Σ. Then, we can compute, in linear time, a linear formula $\Phi_{\mathbb{P}}(A)$, over $\#\Sigma$, such that $[\![\Phi_{\mathbb{P}}(A)]\!]_{\#\Sigma} = \mathbb{P}(L(A))$.*

3 String Constraints

In this section, we formally define string constraints. To begin with, we fix a finite alphabet $\Sigma \subseteq \mathbb{N}$. Note that here we assume that the alphabet is a finite subset of natural numbers. Essentially, we try to capture the numerical encoding of the corresponding symbols in computers (e.g., in ASCII, 'A' is encoded as 65). Hence, we can assume w.l.o.g. that there is a one-to-one mapping between numbers in Σ and the

character it encodes. For the simplicity of presentation, we assume that the character '0' is mapped to the number 0, '1' to 1,. . ., and '9' to 9. For other character c, we use $[\![c]\!]$ to denote the number that it maps to. Notice that this approach is general enough to support any finite set of characters.

A minor technical difficulty is that sometimes we may need to treat ϵ as a number. Therefore, we encode ϵ as some fixed number $[\![\epsilon]\!] \in \mathbb{N} \setminus \Sigma$.

Assume that X is a set of *string variables* ranging over Σ^* and Z a set of *integer variables* ranging over $\mathbb{Z}$. An *interpretation over X and Z* is a mapping $I : X \cup Z \to \Sigma^* \cup \mathbb{Z}$. A *word term* is an element in X^*. We lift the interpretation I to word terms and linear constraints in the standard manner.

We use four types of *atomic string constraint*:

- An *equality constraint* ϕ_e is of the form $t_1 = t_2$ where t_1, t_2 are word terms. The *model* of ϕ_e is the set of interpretations $[\![\phi_e]\!] = \{I \mid I(t_1) = I(t_2)\}$. A *disequality constraint* ϕ_d is of the form $t_1 \neq t_2$ and is interpreted analogously.
- An *integer constraint* ϕ_i is a linear constraint over the integer variables in Z and values of $|x|$ for all $x \in X$, where $|\cdot| : X \to \mathbb{N}$ is the string length function defined in the standard way. We define $[\![\phi_i]\!] = \{I \mid I(\phi_i) = \text{true}\}$.
- A *regular constraint* ϕ_r is of the form $x \in L(A)$ where x is a string variable and A is a finite automaton. The *model* of ϕ_m is the set of interpretations $[\![\phi_m]\!] = \{I \mid I(x) \in L(A)\}$.
- A *string-number conversion constraint* ϕ_s is of the form $n = \mathsf{toNum}(x)$, where the function $\mathsf{toNum}(x)$ is defined as follows. For $a \in [0, 9]$, we have $\mathsf{toNum}(a) = a$ and for $w \cdot a \in [0, 9]^+$, $\mathsf{toNum}(w \cdot a) = 10 \times \mathsf{toNum}(w) + a$. For $w \notin [0, 9]^+$, $\mathsf{toNum}(w) = -1$. We define $[\![\phi_s]\!] = \{I \mid I(n) = \mathsf{toNum}(I(x))\}$. The *number-string* conversion constraint $x = \mathsf{toStr}(n)$ is treated as a syntactic sugar for $n = \mathsf{toNum}(x)$. We assume decimal encoding of numbers.

A *string constraint* is then a conjunction of atomic string constraints, with the semantics defined in the standard manner. It is *satisfiable* if there is an interpretation which evaluates the constraint to true. Often we refer to the first three types of atomic string constraints the *basic string constraints*.

Notice that only positive integer is supported in the string-number conversion function. This is the semantics used by most of the SMT solvers, and hence we follow it in this paper. The encoding has a benefit that it can also handle the case where x is "not a number", using the condition $\mathsf{toNum}(x) = -1$. Supporting only positive integer is not a strong restriction, since converting from negative integer can still be encoded using only the positive version.

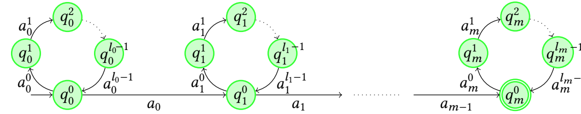
4 Decision Procedure Overview

Our decision procedure has two steps: The first step consists in over-approximating the set of input constraints into a set that falls in the chain-free fragment [5], which is decidable. The over-approximation module proceeds as follows: First, it replaces all string-number conversion constraint $n = \mathsf{toNum}(x)$ by $n = -1 \vee (n \neq -1 \wedge x \in [0-9]^*)$ to obtain an over-approximation Φ consisting of only basic string constraints. Then, it over-approximates Φ to a *chain-free string constraint* [5], which consists of only integer, membership, and *chain-free* equality constraints. Informally, a set of equality constraints has a *chain* if we can find some circular dependency between the string variables in the equality constraints. Our procedure iteratively searches for such dependency chains in the equality constraints. If a chain is found then we replace a variable appearing in that chain by a fresh one. By doing this, we break that chain. We repeat this procedure until there are no more chains. Observe that if the over-approximation is UNSAT then our decision procedure declares that the original formula is also UNSAT.

The second step of our decision procedure is only enabled if the over-approximation step returns SAT. In this case, our decision procedure under-approximates the string constraints by restricting the search domain of each string variable to the language defined by some PFA. This approach based on PFA allows very efficient manipulation. We will show that given such restriction, one can reduce the string constraint solving problem to a linear formula satisfiability problem. The rest of the paper will be mainly dedicated to the explanation of the under-approximation technique (which is our main contribution).

5 Parametric Flat Automata

We introduce *parametric flat automata* that will be used to define patterns used by the under-approximation module to restrict the domain of string variables.



Flat Automata. A finite state automaton $A = (Q, T, \Sigma, q_0^0, q_m^0)$ is said to be *flat* if it satisfies the following structural constraints (see also the figure above):

1. The final state q_m^0 is reached from the initial state q_0^0 through a straight path of $m-1$ transitions $(q_i^0, a_i, q_{i+1}^0) \in T$ with $q_i^0 \in Q$ and $a_i \in \Sigma$ for $i \in [0, m-1]$.
2. Each state q_i^0 is the origin of a unique simple cycle of the length $l_i \in \mathbb{N}$, consisting of states $q_i^{j-1} \in Q$ and transitions $(q_i^{j-1}, a_i^{j-1}, q_i^{j \bmod l_i})$, with $a_i^{j-1} \in \Sigma$, for $j \in [1, l_i]$. Notice that the case when $l_i = 0$ is also admissible and means that there is no cycle on q_i.

3. Each character in Σ appears on at most one transition of the automaton A.

The crucial feature of flat automata is that their semantics can be faithfully represented by a linear formula and handled efficiently by an SMT solver. Such encoding into linear formula results in efficient algorithms and decision procedures. For instance, we avoid dealing with costly standard automata operations (e.g., checking the non-emptiness of the intersection of several regular languages is known to be PSPACE-complete while it is in NP for the class of flat automata). The encoding is possible due to the flat structure, which has the property that "every word $w \in L(A)$ is uniquely determined by its Parikh image $\mathbb{P}(w)$". More precisely, the Parikh image of a word $w \in L(A)$ can be seen as an encoding of w and can be uniquely decoded:

Lemma 5.1. *For a flat FA A, there is a function $decode_A$ such that for each $w \in L(A)$, $decode_A(\mathbb{P}(w)) = w$.*

Observe that the $\mathbb{P}(w)$ value of any variable appearing within a cycle of A is equal to the number of repetitions of that cycle in the accepting run. This is an immediate consequence of the fact that every character appears on at most one transition. Thus, the accepting run on w (and so w itself) can be reconstructed from $\mathbb{P}(w)$.

More concretely, the function $decode_A$ can be implemented as follows. Given $I_\# : \#\Sigma \to \mathbb{N}$, and assuming that the lengths of the loops of A are $l_0, \ldots, l_m$, $decode_A(I_\#)$ is constructed as the word $w_0 a_0 w_1 \cdots a_{m-1} w_m$ where for each $i \in [0, m]$, $w_i = (a_i^0 \cdots a_i^{l_i - 1})^{\# a_i^0}$ if $l_i > 0$ and $w_i = \epsilon$ if $l_i = 0$.

For example, in the automaton given at the beginning of this section, from $|x|_{a_0} = |x|_{a_1} = \cdots = |x|_{a_{m-1}} = 1$, $|x|_{a_1^0} = |x|_{a_1^1} = |x|_{a_1^2} = 2$ and $|x|_{a_j^i} = 0$ otherwise, we derive that $x = a_0(a_1^0 a_1^1 a_1^2)^2 a_1 \cdots a_{m-1}$.

Parametric (Flat) Automata. Next, we define *parametric automaton* (PA) as a pair $P = (A, \psi)$ where A is an automaton operating over an alphabet V of *character variables* and ψ is an *interpretation constraint*, a linear formula over V. *Parametric flat automaton (PFA)* is then a parametric automaton whose automaton is flat. See Figure 1 and 2 for examples of PFAs (without interpretation constraints, i.e., ψ = true.).

Parametric automata accept words over V, called *parametric words*, but we still use them as representations of languages over Σ. Namely, words over V are interpreted as words over Σ under an *interpretation of* V, a mapping $I : V \to \Sigma_\epsilon$ (recall that $\Sigma_\epsilon ::= \Sigma \cup \{\epsilon\} \subseteq \mathbb{N}$). For a parametric word $x = v_1 v_2 \cdots v_k$ over V, its interpretation $I(x)$ is then defined as $I(v_1) \cdot I(v_2) \cdot \ldots \cdot I(v_k)$. We then define the semantics of the PA P as the set of strings $[\![P]\!] = \{I(x) \mid x \in L(A), I \in [\![\psi]\!]\}$ of all interpretations satisfying ψ of all parametric strings in the language of A.

We say that a mapping $I_e : V \cup \#V \to \mathbb{N}$ is a *word encoding* of a word w (or a P-encoding of w) if w is an instantiation of some parametric word $x \in L(P)$ whose Parikh image and interpretation of character variables are defined by I_e. Conversely, w is a *P-decoding* of I_e. We use $encode_P(w)$ below to denote all P-encodings of a word w, and $decode_P(I_e)$ to denote all P-decodings of a word encoding I_e. Namely,

$$encode_P(w) = \{I_e | x \in L(P), I(x) = w, I \in [\![\psi]\!], I_e = I \cup \mathbb{P}(x)\}$$
$$decode_P(I_e) = \{w | x \in L(P), I(x) = w, I \in [\![\psi]\!], I_e = I \cup \mathbb{P}(x)\}$$

Since a word encoding I_e only records the numbers of occurrences of character variables (Parikh image), the same word encoding may be shared by multiple words, as formalized in the definition of $decode_P(I_e)$.

Example 5.2. Let use consider the PFA $P_x = (A_x, \text{true})$ from Figure 2 (a) and let $Y = (v_1, v_2, \#v_1, \#v_2)$. Then we have $encode_{P_x}(\text{"}aaa\text{"}) = \{(Y \to ([\![a]\!], [\![\epsilon]\!], 3, 3), Y \to ([\![\epsilon]\!], [\![a]\!], 3, 3)\}$ and $decode_{P_x}((Y \to ([\![a]\!], [\![\epsilon]\!], 3, 3)) = \{\text{"}aaa\text{"}\}$.

If P is a PFA, then by Lemma 5.1, every word encoding $I_e \in encode_P(w)$ can be decoded uniquely to the word w, i.e.

$$\{w\} = decode_P(encode_P(w))$$

Similarly, as stated by the following corollary of Lemma 5.1, Parikh image of parametric words in $L(A)$ paired with character variable interpretations satisfying ψ encode precisely the words in $[\![P]\!]$.

Corollary 5.3. *For a PFA $P = (A, \psi)$,*
$$[\![P]\!] = decode_P(\{(I \cup I_\#) \mid I_\# \in \mathbb{P}(L(A)), I \in [\![\psi]\!]\}).$$

6 Flat Domain Restriction

In this section, we describe formally how to restrict the domain of string variables to patterns defined by PFA. We start the description of the algorithm that converts a string constraint ϕ_{in} to a linear formula representing the set of solutions under the domain restriction.

The domain restriction is formally defined by restricting the domain of each string variable by a chosen PFA. Namely, assuming that X is the set of string variables of ϕ_{in}, a *flat domain restriction* for ϕ_{in} is a mapping $\mathcal{R}$ that assigns to each variable $x \in X$, a PFA $\mathcal{R}(x)$ over character variables V_x. Let $V_\mathcal{R} = \bigcup_{x \in X} V_x$ be the set of all character variables used in $\mathcal{R}$. We require that these PFA operate over pairwise disjoint sets of character variables, that is if $x \neq y$ then $V_x \cap V_y = \emptyset$. The particular choice of a PFA for each variable depends on the strategy used in the implementation, and will be discussed in Section 9. The *flattening* of the input string constraint ϕ_{in}, denoted $flatten_\mathcal{R}(\phi_{in})$, will be built inductively following the structure of ϕ_{in}. For a conjunction of string constraints, we let $flatten_\mathcal{R}(\phi \wedge \phi') ::= flatten_\mathcal{R}(\phi) \wedge flatten_\mathcal{R}(\phi')$. We do such decomposition until reached atomic string constraints. We show how to build a flattening $flatten_\mathcal{R}(\phi)$ for every atomic string constraint ϕ in the following sections.

The semantics of a string constraint ϕ restricted by $\mathcal{R}$ is then defined as $[\![\phi]\!]^\mathcal{R} = \{I \in [\![\phi]\!] \mid \forall x \in X : I(x) \in [\![\mathcal{R}(x)]\!]\}$.

The correctness of the entire construction of $flatten_\mathcal{R}(\phi_{in})$ is expressed by Theorem 6.2. It uses the decoding function

$decode_{\mathcal{R}}$ parameterized by the domain restriction $\mathcal{R}$. Let Z be the set of integer variables in ϕ_{in}. The function maps an interpretations I_e over $Z \cup V_{\mathcal{R}} \cup \#V_{\mathcal{R}}$ to an interpretation over $Z \cup X$, following the domain restriction $\mathcal{R}$. Informally, it "decodes" an interpretation of integer variables $Z \cup V_{\mathcal{R}} \cup \#V_{\mathcal{R}}$ to an interpretation of variables in the string constraint ϕ_{in}. Formally, we define $decode_{\mathcal{R}}(I_e) ::= \{I \mid \forall z \in Z : I(z) = I_e(z) \wedge \forall x \in X : \{I(x)\} = decode_{\mathcal{R}(x)}((I_e)_{V_{\mathcal{R}(x)} \cup \# V_{\mathcal{R}(x)}})\}$. The condition says that (1) I and I_e are consistent over variables in Z and (2) $(I_e)_{V_{\mathcal{R}(x)} \cup \# V_{\mathcal{R}(x)}}$ is a word encoding that $\mathcal{R}(x)$-encodes $I(x)$. We also define the $\mathcal{R}$-encoding function as the counterpart of $\mathcal{R}$-decoding, namely, for a interpretation I of the string constraint ϕ_{in}, we let $encode_{\mathcal{R}}(I) = \{I_e \mid decode_{\mathcal{R}}(I_e) = \{I\}\}$. We lift $decode_{\mathcal{R}}$ and $encode_{\mathcal{R}}$ to sets of interpretations in the standard manner.

Example 6.1. We consider the domain restriction $\mathcal{R}$ such that $\mathcal{R}(x) = (A_x, \text{true})$ from Figure 2 (a) and $\mathcal{R}(y) = (A_y, \text{true})$ from Figure 2 (b). Let the set of integer variables be $Z = \{v_z\}$ and let $V_{\mathcal{R}(x)} = \{v_0, v_1, v_2, v_3\}$. For the interpretation $I_e = (v_z, v_0, v_1, v_2, v_3, \#v_0, \#v_1, \#v_2, \#v_3) \rightarrow (3, [\![a]\!], [\![b]\!], [\![c]\!], [\![\epsilon]\!], 3, 3, 2, 2)$, we have $decode_{\mathcal{R}}(I_e) = \{(z, x, y) \rightarrow (3, \text{“}ababab\text{”}, \text{“}cc\text{”})\}$.
Conversely, for $I = (z, x, y) \rightarrow (3, \text{“}ababab\text{”}, \text{“}cc\text{”})$ and $Y = (v_z, v_0, v_1, v_2, v_3, \#v_0, \#v_1, \#v_2, \#v_3)$, we have $encode_{\mathcal{R}}(I) =$

$$\left\{ \begin{array}{l} Y \rightarrow (3, [\![a]\!], [\![b]\!], [\![c]\!], [\![\epsilon]\!], 3, 3, 2, 2), \\ Y \rightarrow (3, [\![a]\!], [\![b]\!], [\![\epsilon]\!], [\![c]\!], 3, 3, 2, 2), \\ Y \rightarrow (3, [\![a]\!], [\![b]\!], [\![c]\!], [\![c]\!], 3, 3, 1, 1) \end{array} \right\}$$

Theorem 6.2. $decode_{\mathcal{R}}([\![flatten_{\mathcal{R}}(\phi_{in})]\!]) = [\![\phi_{in}]\!]^{\mathcal{R}}$

The theorem can be proved by a structural induction over ϕ_{in}. However, for the induction step to go through, we will need to guarantee a stronger correspondence of string constraints ϕ and their flattenings $flatten_{\mathcal{R}}(\phi)$ than just the semantic equality $decode_{\mathcal{R}}([\![flatten_{\mathcal{R}}(\phi)]\!]) = [\![\phi]\!]^{\mathcal{R}}$. Particularly, we will need to ensure that $flatten_{\mathcal{R}}(\phi)$ captures exactly all $\mathcal{R}$-encodings of $[\![\phi]\!]^{\mathcal{R}}$ (indeed, notice that if it would be allowed to capture only some of the encodings, then for instance $flatten_{\mathcal{R}}(\phi) \wedge flatten_{\mathcal{R}}(\phi')$ could only underapproximate $[\![\phi \wedge \phi']\!]^{\mathcal{R}}$). The inductive argument needed in the correctness proof of the under-approximation then reads as

$$[\![flatten_{\mathcal{R}}(\phi)]\!]_{V_{\mathcal{R}} \cup V_{\#\mathcal{R}}} = encode_{\mathcal{R}}[\![\phi]\!]$$

In the next sections, we will formulate the corresponding correctness lemma for flattening constructed from each type of atomic string constraints. We note that the restriction of $[\![flatten_{\mathcal{R}}(\phi)]\!]$ to $V_{\mathcal{R}} \cup V_{\#\mathcal{R}}$ here is needed since $flatten_{\mathcal{R}}(\phi)$ will be constructed with some auxiliary variables.

7 Flattening of Basic String Constraints

We will first discuss flattening of the basic string constraints, that is, regular, equality, and integer constraints. We start by two needed operations over PA, synchronization and concatenation.

Synchronization of PAs. We will now discuss a construction of the *synchronization formula* for two PAs P and P'. It is a linear formula $\Psi_{P \times P'}$ that specifies how each word in the semantic intersection $[\![P]\!] \cap [\![P']\!]$ is encoded by P and by P'. More precisely, the models of $\Psi_{P \times P'}$ represent pairs of word encodings I_e and I'_e such that $I_e \in encode_P(w)$ and $I'_e \in encode_{P'}(w)$ (hence $w \in [\![P]\!] \cap [\![P']\!]$).

Particularly, the synchronization formula is built for two PAs $P=((Q, T, V, q_i, q_f), \psi)$ and $P'=((Q', T', V', q'_i, q'_f), \psi')$ such that $V \cap V' = \emptyset$. It is extracted from the *asynchronous product* of P and P'. The asynchronous product is an automaton that uses $Q \times Q'$ as the set of states and $V_\epsilon \times V'_\epsilon$ as the alphabet. Every accepting run of $P \times P'$ corresponds to a pair of accepting runs, a run of P over a parametric word x and a run of P' over a parametric word x'. The word accepted by the run of $P \times P'$ induces constraints on the interpretations of I over V and I' over V' under which the two parametric words have the same interpretation, i.e. $I(x) = I'(x')$.

Intuitively, when the product automaton $P \times P'$ takes a transition $((q_1, q'_1), (v, v'), (q_2, q'_2))$, it means the character variable v and v' should be assigned the same value, P moves under v from state q_1 to state q_2 and P' from q'_1 to q'_2 under v'. When $P \times P'$ takes a transition $((q_1, q'_1), (v, \epsilon), (q_2, q'_1))$, it means that the character variable v should be assigned ϵ, P moves under v to q_1, and P' takes no action, since no action is needed to match P's reading of ϵ (hence consumes no symbol from the input word). Symmetrically, P' might read a variable v assigned ϵ and P may stay.

Formally, the asynchronous product automaton is a tuple $P \times P' = (Q \times Q', T_\times, V_\epsilon \times V'_\epsilon, (q_i, q'_i), (q_f, q'_f))$, where the transition relation $T_\times$ is the minimal set satisfying the following:

- If $(q_1, v, q_2) \in T$ and $(q'_1, v', q'_2) \in T'$, then we have $((q_1, q'_1), (v, v'), (q_2, q'_2)) \in T_\times$.
- If $(q_1, v, q_2) \in T$, then for all states $q' \in Q'$, we have $((q_1, q'), (v, \epsilon), (q_2, q')) \in T_\times$.
- If $(q'_1, v', q'_2) \in T'$, then for all states $q \in Q$, we have $((q, q'_1), (\epsilon, v'), (q, q'_2)) \in T_\times$.

The synchronization formula $\Psi_{P \times P'}$ is extracted from $P \times P'$ as follows. Its first part is the Parikh formula $\Phi_{\mathbb{P}}(P \times P')$ of the product, which encodes all accepting runs of $P \times P'$. The second part is a constraint that extracts from a run of $P \times P'$ the corresponding runs of P and of P':

$$\Psi_{\#} ::= \left(\bigwedge_{v \in V} \#v = \sum_{x' \in V'_\epsilon} \#(v, x') \right) \wedge \left(\bigwedge_{v' \in V'} \#v' = \sum_{x \in V_\epsilon} \#(x, v') \right)$$

Notice that x, x' are either variables or ϵ. Finally, the third part forces the interpretations of the parametric words accepted by P and P' to be the same:

$$\Psi_{=} ::= \bigwedge_{x \in V_\epsilon, x' \in V'_\epsilon} \#(x, x') > 0 \rightarrow (x = x')$$

The synchronization formula is then the conjunction

$$\Psi_{P\times P'} ::= \Phi_{\mathbb{P}}(P \times P') \wedge \Psi_{\#} \wedge \Psi_{=} \wedge \psi \wedge \psi'$$

The correctness of this construction is stated in Lemma 7.1 below. The correctness of the construction of under-approximations of equality constraints and regular constraints in Section 7.1 and Section 7.2 rely on it.

Lemma 7.1. $[\![\Psi_{P\times P'}]\!]_{V\cup V'\cup\#V\cup\#V'} = \{I_e \cup I'_e \mid I_e \in encode_P(w), I'_e \in encode_{P'}(w), w \in [\![P]\!] \cap [\![P']\!]\}$

Informally, the lemma states that the models of $\Psi_{P\times P'}$ encode precisely the pairs of equivalent encodings of words from $[\![P]\!]$ and $[\![P']\!]$, that constitute the intersection $[\![P]\!] \cap [\![P']\!]$. Since that the models of $\Psi_{P\times P'}$ include also an assignment to the auxiliary variables of $(V_\epsilon \times V'_\epsilon) \cup \#(V_\epsilon \times V'_\epsilon)$, the lemma restricts $[\![\Psi_{P\times P'}]\!]$ to $V \cup V' \cup \#V \cup \#V'$.

Notice that if P is flat (or, symmetrically, if P' is flat), then the semantic intersection $[\![P]\!] \cap [\![P']\!]$ can be still decoded from the synchronization formula $\Psi_{P\times P'}$. Namely, due to Corollary 5.3, we have that if P is a PFA, then

$$decode_P([\![\Psi_{P\times P'}]\!]_{V\cup\#V}) = [\![P]\!] \cap [\![P']\!]$$

Concatenation of PFAs. Concatenation of PFAs will be needed when flattening equality constraints. Its implementation is straightforward, connect the final state of the first PFA with the initial state of the second by an ϵ-transition. Since our automata do not allow transition directly labeled by ϵ, the ϵ-transition is labeled by a fresh variable v_ϵ forced by the constraint $v_\epsilon = \epsilon$ to take the value ϵ.

Formally, given PFA $P = ((Q, T, V, q_i, q_f), \psi)$ and $P' = ((Q', T', V', q'_i, q'_f), \psi')$ with $Q \cap Q' = \emptyset = V \cap V'$, their concatenation is the PFA $P \cdot P' = (Q \cup Q', T \cup T' \cup \{(q_f, v_\epsilon, q'_i)\}, V \cup V' \cup \{v_\epsilon\}, q_i, q'_f, \psi \wedge \psi' \wedge v_\epsilon = \epsilon)$ where v_ϵ is fresh, not from $V \cup V'$.

Lemma 7.2. *$encode_{P\cdot P'}([\![P \cdot P]\!])_{V\cup V'\cup\#V\cup\#V'} = \{I_e \cup I'_e \mid I_e \in encode_P([\![P]\!]) \wedge I'_e \in encode_{P'}([\![P']\!])\}$, for PFAs P and P'.*

With synchronization and concatenation of PA, we are ready to describe flattening of the basic string constraints.

7.1 Flattening of Regular Constraints

Let us first describe the construction of $flatten_{\mathcal{R}}(\phi_r)$ for a regular constraint $\phi_r ::= x \in L(A)$. In order to synchronize the FA A with the PFA $\mathcal{R}(x)$, we represent A by a PA $P' = (A', \Psi_{char})$. The automaton A' of P' operates over fresh character variables $v_a, a \in \Sigma_\epsilon$, and is obtained from A by replacing every occurrence of each character $a \in \Sigma_\epsilon$ on a transition by the variable v_a. The interpretation restriction formula Ψ_{char} of P' then binds the fresh character variables to the character values they represent, namely, $\Psi_{char} = \bigwedge_{a\in\Sigma_\epsilon} v_a = [\![a]\!]$. Obviously, $L(A) = [\![P']\!]$. We then let

$$flatten_{\mathcal{R}}(\phi_r) = \Psi_{\mathcal{R}(x)\times P'}$$

The following lemma states the correctness of this construction. It follows from Corollary 5.3 and Lemma 7.1.

Lemma 7.3. $[\![flatten_{\mathcal{R}}(\phi_r)]\!]_{V_{\mathcal{R}}\cup\#V_{\mathcal{R}}} = encode_{\mathcal{R}}([\![\phi_r]\!])$.

7.2 Flattening of Equality Constraints

We now describe the construction of $flatten_{\mathcal{R}}(\phi_e)$ for an equality constraint $\phi_e ::= x_0 \cdot x_1 \cdots x_n = x_{n+1} \cdot x_{n+2} \cdots x_m$. To simplify the presentation, we assume that the variables are pairwise different, i.e. that $i \neq j \implies x_i \neq x_j$. We may make this assumption without loss of generality, since multiple occurrences of variables in ϕ_e can be eliminated. In fact, whenever $x_i = x_j$ for $i \neq j$, we may replace x_j by a fresh variable x'_j and conjoin the modified ϕ_e with a new equality $x_j = x'_j$. We also assume that all disequality constraint $t \neq t'$ are already converted to equivalent equality constraints and integer constraints in the standard way [4].

Having made these assumptions, we may proceed follows. First, we build two PFAs P^{left} and P^{right} that encode the left and the right-hand side word term of the equality constraint ϕ_e, respectively, by concatenating the restrictions of the individual variables. That is

$$P^{left} ::= \mathcal{R}(x_1) \cdot \ldots \cdot \mathcal{R}(x_n) \quad P^{right} ::= \mathcal{R}(x_{n+1}) \cdot \ldots \cdot \mathcal{R}(x_m)$$

The under-approximation of ϕ_e is then obtained as their synchronization formula

$$flatten_{\mathcal{R}}(\phi_e) = \Psi_{P^{left}\times P^{right}}$$

The correctness of the construction is stated by the lemma:

Lemma 7.4. $[\![flatten_{\mathcal{R}}(\phi_e)]\!]_{V_{\mathcal{R}}\cup\#V_{\mathcal{R}}} = encode_{\mathcal{R}}([\![\phi_e]\!])$.

7.3 Flattening of Integer Constraints

Given an integer constraint ϕ_l that talks about lengths $|x|$ of string variables $x \in X$. We use a version of ϕ_l where every occurrence of every length function $|x|$ is replaced by an auxiliary length variable l_x and we add a formula to ensure that the value of l_x is equal to that of $|x|$ even when x is encoded using the character variables V_x and Parikh variables $\#V_x$ of $\mathcal{R}(x)$. We will need a set auxiliary variables $\{l_v \mid v \in V_x\}$ to express the length by which the character variable v contributes to the length of an $\mathcal{R}$-encoded string x. That is, l_v will be 0 if v is assigned $[\![\epsilon]\!]$, otherwise it will equal to the number $\#v$ of its occurrences in the word:

$$\Psi_{l_v} ::= (v = [\![\epsilon]\!] \wedge l_v = 0) \vee (v \neq [\![\epsilon]\!] \wedge l_v = \#v)$$

The length of the encoded word x is then the sum of the lengths contributed by the individual character variables in V_x, hence we let

$$\Psi_{l_x} ::= l_x = \sum_{v\in V_x} l_v \wedge \bigwedge_{v\in V_x} \Psi_{l_v} .$$

Finally, the linear formula created for ϕ_l is

$$flatten_{\mathcal{R}}(\phi_l) ::= \phi_l \wedge \bigwedge_{x\in X} \Psi_{l_x}$$

and the following lemma states its correctness.

Lemma 7.5. $[\![flatten_{\mathcal{R}}(\phi_l)]\!]_{V_{\mathcal{R}}\cup\#V_{\mathcal{R}}} = encode_{\mathcal{R}}([\![\phi_l]\!])$

8 Flattening of String-Number Conversion

Last, we present the main contribution of this paper, the construction of a flattening $flatten_{\mathcal{R}}(\phi_s)$ of string-number conversion constraint $\phi_s ::= n = \mathsf{toNum}(x)$.

Let us begin with a simple example. Assume that we use the PFA in Figure 2 (a) to restrict the domain of x. Then we know that when $0 \le v_0, v_1 \le 9$, then n is a positive integer value, and otherwise $n = -1$. So we should first add the constraint $((0 \le v_0 \le 9) \land (0 \le v_1 \le 9)) \lor (n = -1 \land (v_0 > 9 \lor v_1 > 9))$.

For the case that n is a positive integer, the value of n can be characterized by a constraint (assume character variables are not assigned ϵ) $n = (v_0 \times 10 + v_1) \times (1 + 100 + 100^2 + \ldots 100^{\#v_0 - 1}) = (v_0 \times 10 + v_1) \times \frac{100^{\#v_0} - 1}{100 - 1}$. The constraint uses $\#v_0$ to capture the total number of loop traversals. Notice that the constraint above contains an exponential component $\frac{100^{\#v_0}}{100-1}$. To solve the satisfiability of this formula, one needs to solve an exponential constraint.

Let us have a look at another example. If we restrict the domain of x to the PFA in Figure 2 (c), for the case when n is positive, we have the relation $n = (v_0 \times 10 + v_1) \times (1 + 100 + 100^2 + \ldots 100^{\#v_0 - 1}) \times 10 \times 100^{\#v_3} + v_2 \times 100^{\#v_3} + (v_3 \times 10 + v_4) \times (1 + 100 + 100^2 + \ldots 100^{\#v_3 - 1}) = (v_0 \times 10 + v_1) \times \frac{100^{\#v_0 - 1}}{100-1} \times 10 \times 100^{\#v_3} + v_2 \times 100^{\#v_3} + (v_3 \times 10 + v_4) \times \frac{100^{\#v_3 - 1}}{100-1}$. Observe that the formula has multiple exponential components, including $\frac{100^{\#v_0} \times 100^{\#v_3}}{100-1}$ and $\frac{100^{\#v_3}}{100-1}$.

It is not difficult to see from the examples above that, if $\mathcal{R}(x)$ is an arbitrary PFA with m loops, the formula that defines the number n contains at least m exponential components, one for each loop. To the best of our knowledge, the satisfiability problem of integer constraints with a mix of polynomials and exponentials is still open. The problem is difficult even for the case that variables are real numbers [17, 18]. For example, the algorithm in [24] involves a quantifier elimination procedure which is double-exponential to the length of the input formula and hence cannot handle large instances. We therefore do not expect that such constraint can be solved efficiently. Instead, we will define a special form of the flat restriction $\mathcal{R}(x)$ of x that leads to an easier linear formula.

Efficient String-Number Conversion using PFA. We will now discuss the special form of $\mathcal{R}(x)$, called *numeric PFA*, which we choose for string variables appearing in string-integer constraints and that leads to efficient under-approximation technique. Besides simple induced linear formulae, we still want single $\mathcal{R}(x)$ to cover as many numerals as possible. We want an "easy" completeness property, that is, (1) the space of all numerals can be covered completely by our special numeric PFAs, (2) these numeric PFAs are generated easily, and (3) each of them covers a large and practically significant portion of numerals (so that satisfiable assignments can be often found within the domain restriction of just few of numeric PFAs). Particularly, we will proceed towards a definition of *numeric PFA* $(A^m, \psi^m), m \in \mathbb{N}$ which covers all numerals that encode integers with m digits.

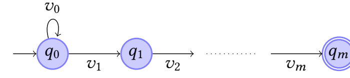


Figure 3. PFA for string-number conversion constraints

Our first attempt is a PFA without loop, i.e., a straight line structure $\rightarrow q_0 \xrightarrow{v_1} q_1 \xrightarrow{v_2} \cdots \xrightarrow{v_m} q_m$. The corresponding integer constraint does not have exponential components. However, it does cover all numbers with at most m-digits. Consider the example $\mathsf{toNum}(x) = 10 \land |x| = 5$. The number 10 has only 2-digits, at the first glance, a straight-line PFA with two transitions, i.e., the PFA $\rightarrow q_0 \xrightarrow{v_1} q_1 \xrightarrow{v_2} q_2$ should be sufficient for the domain restriction of x. If we do so, we will conclude that the formula is unsatisfiable, because the length of x cannot be 5 under this domain restriction.

However, the formula is satisfiable when x = "00010". Observe that toNum("00010") = 10 since PFAs accept numerals with the least significant bit first. The key is that even for a bounded integer, the corresponding numeral can be of an unbounded length with arbitrarily many trailing '0's at the front. All numbers with up to m-digits can be however still handled without having to solve exponential constraints. It is enough to equip the initial state of the PFA with a 0-self-loop.

Consequently, the automaton A^m of our numeric PFA will have the following form illustrated in Figure 3. It has a self-loop on the initial state labeled by the character variable v_0, forced by the constraint

$$\Psi_{v_0} ::= v_0 = 0$$

to hold the value 0. This transition ensures that the under-approximation handles numerals with arbitrary number of trailing zeros. The self-loop is followed by a chain of m transitions $(q_{i-1}, v_i, q_i), 1 \le i \le m$, leading towards the final state q_m. The chain encodes at most m meaningful digits (only *at most* because the first variables in the sequence can still be assigned zeros and some variables may be assigned ϵ). Hence this PFA covers all numerals that encode numbers with at most m digits. Although it still has a loop, it will not create any exponential component defining the value of n because the loop only represents a sequence of "0" at the front of x. Thus, it will not affect the integer value of $n = \mathsf{toNum}(x)$.

Numeric PFA with these restrictions would satisfy our primary objective, that is, they would induce linear formulae and would "easily" and completely cover all numerals. A last problem still needs to be solved before they can be efficient in practice. Recall that the character variables can be assigned ϵ. Therefore, a single chain of k interesting digits, $k \le m$, can be by A^m represented in $\binom{k}{m}$ ways, each corresponding to one possible interleaving of k digits with $m-k$ epsilons. This may

lead to a formula of exponential size when defining the value of n. In order to eliminate this potential blow-up in the size of the formula, we add to ψ^m an additional constraint that forces all epsilons to be *shifted* behind the least significant digit. This will leave us with only one interleaving. This is the formula

$$\Psi^m_{\text{shift}} ::= \bigwedge_{1\leq i\leq m} v_i \neq [\![\epsilon]\!] \implies v_{i-1} \neq [\![\epsilon]\!] \ .$$

Last, since this restriction is meaningful only when the string is indeed a numeral, we also define the constraint representing the strings which are not numerals, the formula

$$\Psi^m_{\text{NaN}} ::= \bigvee_{i\in[1,m]} v_i > 9$$

and define the final form of the interpretation restriction used by A^m as

$$\psi^m ::= \Psi^m_{\text{NaN}} \vee (\Psi_{v_0} \wedge \Psi^m_{\text{shift}}) \ .$$

Consequently, we design our domain restrictions $\mathcal{R}$ so that for string variables x that appear within string-integer constraints, $\mathcal{R}(x)$ is a numeric PFA (A^m, ψ^m), $m \in \mathbb{N}$.

Assuming that the domain restriction for x is (A^m, ψ^m), the value of the integer n can be extracted from a numeral using the formula

$$\Psi^m_{\text{toInt}} ::= \bigvee_{1\leq k\leq m} \Psi^m_{\text{last(k)}} \wedge (n = v_1 * 10^{k-1} + v_2 * 10^{k-2} + \ldots + v_k)$$

where $\Psi^m_{\text{last(k)}}$ says that the last variable of A^m assigned a non-ϵ value is v_k, namely

$$\Psi^m_{\text{last(k)}} ::= (k = m \wedge v_k \geq 0) \vee (v_k \geq 0 \wedge v_{k+1} = -1) \ .$$

Since we also need to distinguish the case when x is not a number, in which case n should equal -1, the formula under-approximating ϕ_s is finally constructed as

$$\mathit{flatten}_{\mathcal{R}}(\phi_s) ::= \Phi_{\mathbb{P}}(A^m) \wedge \big((\Psi^m_{\text{NaN}} \wedge n = -1) \vee (\neg\Psi^m_{\text{NaN}} \wedge \Psi^m_{\text{toInt}})\big)$$

The following lemma states correctness of this construction:

Lemma 8.1. $[\![\mathit{flatten}_{\mathcal{R}}(\psi_s)]\!]_{V_{\mathcal{R}}\cup\#V_{\mathcal{R}}} = \mathit{encode}_{\mathcal{R}}([\![\phi_s]\!])$

9 Implementation and Evaluation

We have implemented our string constraint solving procedure in a tool called Z3-Trau. Z3-Trau is implemented as a theory solver of the SMT solver Z3 [15]. In this way, we can concentrate on solving conjunctive constraints and let Z3 handle the other boolean connectives. Secondly, it makes it possible to solve not only formulae over string constraints but also combinations of string constraints with other theories that Z3 supports. Furthermore, this approach allows us to more effectively handle the arithmetic constraints that are generated by the under-approximation module and, lastly, it eliminates the need to have our own parser.

In Z3-Trau, we use the following PFA selection strategy. We use *numeric PFAs* for string variables appearing in string-number conversion and *standard PFAs* for others. We select a size m for numeric PFAs, a number p of their loops, and the length q of the loops. Initially, we set $(m, p, q) = (5, 2, q)$ where q is dynamic and obtained from our internal static analysis. We double m and increase p and q by one if refinement is required. We set an upper bound for each parameter and report UNKNOWN if a solution cannot be found within the bound.

Our over-approximation module also uses heuristics to derive the constant value of any side of the constraint $n = \text{toNum}(x)$ to refine the over-approximation. For instance, assume we can derive that $n = 12$ from some integer constraints. Then we can derive the value of x belongs to the regular language (0^*12).

The way our theory solver and Z3 interact is almost standard. When Z3 asks our theory solver a string constraint satisfiability problem, our solver tries to prove it is SAT or UNSAT using the procedure discussed in this paper. For under-approximation, whenever a corresponding linear formula is created, we attach the current value of m, p, q to the formula, and then push it to Z3 core. If our theory solver reports UNKNOWN, Z3 remembers it in a global flag incomplete and either tries another solution branch, or the same solution branch with different value of m, p, q. If Z3 completes the search of all solution branches, it reports UNSAT if the flag incomplete is down, and UNKNOWN otherwise.

We compare Z3-Trau (1e715b7dab)[2] with other state-of-the-art string solvers, namely, CVC4 (version 1.7) [8], Z3 (version 4.8.7) [15], and Z3Str3 (version 4.8.7) [47]. For these tools, the versions we used are the latest release version. Observe that CVC4 and Z3 are DPLL(T)-based string solvers. We do not compare with Sloth [21] since it does not support length constraints, which occur in most of our benchmarks. We also do not compare with ABC [6] (a model counter for string constraints), Ostrich [13] and Trau+ [5], because they do not support many of the string functions in our benchmarks, especially string-number conversion.

We perform two sets of experiments. In the first set of experiments, we compare Z3-Trau with other tools on existing benchmarks over basic string constraints. Those benchmarks do not involve string-number conversion functions. In the second set of experiments, we compare Z3-Trau with the other tools on new suites focusing on string-number conversion. Our goals of experiments are the following:

- Z3-Trau performs as good as or better than the other tools in solving the satisfiability problems of basic string constraints.
- Z3-Trau performs significantly better than the other tools in solving the satisfiability problems on string-number conversion benchmark, and this shows the

[2] https://github.com/guluchen/z3/tree/1e715b7dab

efficiency of PFA in general and *numeric* PFA in particular.

In the first set of experiments, we use the following benchmark examples:

- PyEx [34] comes from running the symbolic executor PyEx over some Python packages.
- LeetCode comes from running PyEx over a sample code collected from the LeetCode [25] website, including functions that check whether a string is a valid IPv4 or IPv6 address, sum up two binary numbers, check whether an input string is an abbreviation of another input string, and convert a sequence of digits to a string according to a given mapping.
- StringFuzz [10] is generated by the fuzz testing tool of the same name.
- $\text{cvc4}_{\text{pred}}$ and $\text{cvc4}_{\text{term}}$ are obtained from the CVC4 group [33]. These benchmarks contain a small amount of string-number conversion constraints (< 5%).

In the second experiment, we compare with tools supporting string-number conversion on the benchmarks collected from the symbolic executor `Py-Conbyte`[3], which has the supports of string-number conversion. We ran it on several examples collected from the LeetCode platform and from Python core libraries, which involve diverse usages of string-number conversion in Python such as parsing date-time, verifying and restoring IP addresses from strings, etc. We also have examples that encode execution paths of some JavaScript programs (the Luhn algorithm and some array manipulations).

All experiments were executed on a machine with 4-core CPU and 16 GiB RAM. The timeout was set to 10s for each test. We use the results from Z3-Trau, CVC4, and Z3 as the reference answer for the validation of the correctness of the results. Occasionally, two of them report inconsistent answers (one SAT and one UNSAT). To decide which solver is right, we developed a validator. It takes the model I returned from the solver who reported SAT, assigns $I(x)$ to all variables x in the test to obtain a modified test, and re-evaluates the modified test by multiple solvers. If the results from all solvers are consistent, we mark the test SAT or UNSAT according to the results. Otherwise, we manually simplify and inspect the test until we get a conclusive result.

The results of the experiments are summarized in Table 1, Table 2, and Table 3. Rows with heading `SAT`/`UNSAT` show numbers of solved formulae. Rows with heading `UNKNOWN` or `TIMEOUT` indicate the number of instances for which the solver fails to return an answer. `ERROR` means system crashes due to various reasons (usually out of memory). `INCORRECT` shows the number of cases where the tool gives a wrong answer.

[3] https://github.com/spencerwuwu/py-conbyte

Table 1. Results of Z3-Trau, CVC4, Z3, and Z3Str3 on Basic String Constraint benchmarks.

		Z3-Trau	CVC4	Z3	Z3Str3
PyEx	SAT	21377	20350	18492	3037
	UNSAT	3860	3841	3847	3816
	UNKNOWN	0	0	0	7
	TIMEOUT	184	1230	3082	16872
	ERROR	0	0	0	1675
	INCORRECT	0	0	0	14
LeetCode	SAT	877	874	881	661
	UNSAT	1785	1785	1785	1780
	UNKNOWN	0	0	0	122
	TIMEOUT	0	7	0	90
	ERROR	4	0	0	13
	INCORRECT	0	0	0	0
StringFuzz	SAT	515	615	338	505
	UNSAT	301	255	198	192
	UNKNOWN	0	0	0	4
	TIMEOUT	249	195	529	364
	ERROR	0	0	0	0
	INCORRECT	0	0	0	0
$\text{cvc4}_{\text{pred}}$	SAT	13	11	12	8
	UNSAT	822	818	808	774
	UNKNOWN	0	0	0	4
	TIMEOUT	0	6	15	38
	ERROR	0	0	0	11
	INCORRECT	0	0	0	0
$\text{cvc4}_{\text{term}}$	SAT	10	9	7	2
	UNSAT	1032	1026	1022	957
	UNKNOWN	0	0	0	3
	TIMEOUT	3	10	16	58
	ERROR	0	0	0	11
	INCORRECT	0	0	0	14
Total	SAT	22792	21859	19730	4213
	UNSAT	7800	7725	7550	7519
	UNKNOWN	0	0	0	140
	TIMEOUT	436	1448	3642	17422
	ERROR	4	0	0	1710
	INCORRECT	0	0	0	28

From Table 1, we can see that the performance of Z3-Trau is as good as that of the most competitive tools such as CVC4 and Z3 on basic string constraints. In all of the benchmarks, Z3-Trau ranked either the 1st or the 2nd on the number of solved (SAT+UNSAT) cases. On the StringFuzz benchmarks that are SAT, Z3-Trau does not perform as well as the best performing tool. We however do not consider this crucial because these benchmarks are just randomly generated for debugging. On the most important benchmarks, those that come from program analysis, Z3-Trau is comparable to the best performing tool.

From Table 2, we can see that Z3-Trau significantly outperforms all the other tools. The second best tool, Z3, fails on 50 times more examples.

As an addition experiment, we have encoded the check-Luhn algorithm introduced in Section 1 for the cases with 2 to 12 loops (digits). We ran these tests with the timeout set

Table 2. Results of Z3-Trau, CVC4, Z3, and Z3Str3 on String-Number Conversion benchmark.

		Z3-Trau	CVC4	Z3	Z3Str3
Leetcode	SAT	2501	1659	1993	239
	UNSAT	16394	15604	16124	15288
	UNKNOWN	0	0	0	623
	TIMEOUT	32	1664	810	2337
	ERROR	0	0	0	332
	INCORRECT	0	0	0	108
PythonLib	SAT	1922	560	1839	206
	UNSAT	724	666	724	642
	UNKNOWN	0	0	0	45
	TIMEOUT	0	1420	83	1710
	ERROR	0	0	0	41
	INCORRECT	0	0	0	2
JavaScript	SAT	20	3	16	4
	UNSAT	0	0	0	0
	UNKNOWN	0	9	0	0
	TIMEOUT	0	8	4	10
	ERROR	0	0	0	6
	INCORRECT	0	0	0	0
Total	SAT	4443	2222	3848	449
	UNSAT	17118	16270	16848	15930
	UNKNOWN	0	9	0	668
	TIMEOUT	32	3092	897	4057
	ERROR	0	0	0	379
	INCORRECT	0	0	0	110

to 120s. The result is summarized in Table 3. In these tests, Z3-Trau can solve all problems within 1s while CVC4 only returns a model for cases of 2 to 5 loops and Z3Str3 could not solve any of these problems (either TIMEOUT, ERROR, or UNKNOWN). However, Z3 can still solve 5 out of the 11 problems. The behavior of Z3 is not entirely unexpected. All the problems are satisfiable and the solver may be lucky to guess the solution quickly.

Table 3. Comparison of Z3-Trau, CVC4, Z3, and Z3Str3 with checkLuhn problems of 2 to 12 loops.

# of Loops	Z3-Trau	CVC4	Z3	Z3Str3
2	**SAT**(0.27s)	**SAT**(0.17s)	**SAT**(0.11s)	**ERROR**
3	**SAT**(0.29s)	**SAT**(6.94s)	**SAT**(0.13s)	**ERROR**
4	**SAT**(0.37s)	**SAT**(4.92s)	**SAT**(0.24s)	**ERROR**
5	**SAT**(0.39s)	**SAT**(30.86s)	**SAT**(0.13s)	**ERROR**
6	**SAT**(0.41s)	**TIMEOUT**	**TIMEOUT**	**UNKNOWN**
7	**SAT**(0.51s)	**TIMEOUT**	**TIMEOUT**	**ERROR**
8	**SAT**(0.53s)	**TIMEOUT**	**TIMEOUT**	**ERROR**
9	**SAT**(0.63s)	**TIMEOUT**	**SAT**(0.31s)	**ERROR**
10	**SAT**(0.69s)	**TIMEOUT**	**TIMEOUT**	**TIMEOUT**
11	**SAT**(0.71s)	**TIMEOUT**	**TIMEOUT**	**ERROR**
12	**SAT**(0.74s)	**TIMEOUT**	**TIMEOUT**	**ERROR**

10 Related Works

To the best of our knowledge, the study of solving string constraint traces back to 1946, when Quine [32] showed that the first-order theory of string equality constraints (a.k.a. word equations) is undecidable. Makanin achieves a milestone [28] by showing that the class of quantifier-free string equality constraints is decidable. Since then, several works, e.g., [19, 20, 29–31, 35, 39], consider the decidability and complexity of different subclasses of string equality constraints.

Satisfiability of string constraints is a challenging problem. The satisfiability of equality constraints combined with length constraints of the form $|x| = |y|$ is already opened for more than 20 years [11]. Numerous decidable fragments were proposed [3, 5, 7, 12, 13, 27]. Among them, the chain-free fragment [5] used by our over-approximation module is the largest known decidable fragment, which allows us to produce more precise over-approximation and hence solve many UNSAT instances efficiently.

The strong practical motivation led to the rise of several string constraints solvers that concentrate on solving practical problem instances. Several tools handle string constraints assuming a fixed upper bound on the length of strings and translate them to boolean satisfiability problems [23, 37, 38]. Our method, on the other hand, allows analyzing constraints without a length limit and still with some completeness guarantees, i.e., within the language defined by PFAs.

More recently, *DPLL(T)-based* string solvers [2, 4, 8, 9, 13, 21, 41, 42, 46] lift the restriction of strings of bounded length. They usually support a variety of string constraints, including all basic string constraints, and sometimes also regular/rational relations. The typical procedure they used for solving equality constraints is to split them into simpler subcases, in combination with powerful techniques for Boolean reasoning to curb the resulting exponential search space. In contrast, our approach uses a completely different search strategy. We restrict the solution space to some predefined pattern and step-wisely enrich the pattern in use.

The most relevant work to ours is the FA-based approach [2] that projects the solution space of variables to a generalization of flat automata. The main difference is that our approach works fully symbolically, which is enabled through using integer variables as characters and hence PFAs. The use of integer variables as characters allows our approach to handle string-number constraints efficiently – the values and number of occurrences of those variables can be directly converted to numbers in a linear formula.

In principle, FA-based solvers can be extended to support string-number conversion too, but PFA are much more efficient. Extending FA-based approach would require multiple if-then-else statements (e.g., saying that if a transition labeled '0' is taken, then the corresponding number is 0), which introduces a significant amount of additional predicates that the DPLL engine needs to evaluate. We have confirmed this in our preliminary experiment.

PASS [26] uses quantified formulae over arrays with symbolic length to encode string constraints, and a specialized quantifier elimination to solve them. Though it differs from our approach significantly and handling the quantification is expensive, PASS is indeed similar in how finite automata and string-number conversion constraints are translated to

formulae. Our translation is in general more efficient because it uses only linear arithmetics. In contrast, PASS translates string constraints to formulae with quantifier and array predicates. We did not compare with PASS empirically. The authors informed us that PASS is no longer maintained and is not open-source.

A further direction is *automata-based* solvers for analyzing string-manipulated programs. ABC [6] and Stranger [44] soundly over-approximates string constraints using multi-tape automata [45], and outperforms DPLL(T)-based solvers when checking single execution traces, according to some evaluations [22]. People also studied the combination of automata-based algorithms with with model checking algorithms, in particular, IC3/PDR, for more efficient checking of the emptiness for automata [21, 43]. However, many kinds of constraints, including length constraints and word equations, cannot be entirely handled by automata-based solvers.

11 Conclusion and Future Works

In this paper, we report a novel approach for solving string constraints with string-number conversion and implemented it as an open-source tool Z3-Trau. For now, it support basic string constraints, string-number conversion, and also operations that can be encoded to them (e.g., CONTAINS, PREFIXOF). Since Z3-Trau is built inside the SMT solver Z3, we also get the power of processing formulae in the combination of different theories (e.g., array). Hence our tool can support the encoding of a wide range of program expressions. There are several avenues for future works. First, we are planning to integrate it with the JavaScript symbolic executor COSETTE [36]. We believe such integration is feasible. We are also planning to merge Z3-Trau with the main branch of the Z3 solver. For technical development, we think it would be interesting to consider the (symbolic) flattening of an even larger set of string operations, such as the one containing REPLACEALL and SPLIT.

Acknowledgments

This research was partially supported by the Swedish Foundation for Strategic Research, Ministry of Science and Technology Project, Taiwan (no. 106-2221-E-001 -009 -MY3), Ministry of Education, Youth and Sports of Czech Republic (project IT4Innovations excellence in science–LQ1602 and project LL1908), and FIT BUT internal grant FIT-S-20-6427.

References

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

[2] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Bui Phi Diep, Lukás Holík, Ahmed Rezine, and Philipp Rümmer. 2018. Trau: SMT solver for string constraints. In *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018*, Nikolaj Bjørner and Arie Gurfinkel (Eds.). IEEE, 1–5.

[3] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Lukás Holík, Ahmed Rezine, Philipp Rümmer, and Jari Stenman. 2014. String Constraints for Verification. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings (Lecture Notes in Computer Science)*, Armin Biere and Roderick Bloem (Eds.), Vol. 8559. Springer, 150–166.

[4] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Lukás Holík, Ahmed Rezine, Philipp Rümmer, and Jari Stenman. 2015. Norn: An SMT Solver for String Constraints. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I (Lecture Notes in Computer Science)*, Daniel Kroening and Corina S. Pasareanu (Eds.), Vol. 9206. Springer, 462–469.

[5] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bui Phi Diep, Lukás Holík, and Petr Janku. 2019. Chain-Free String Constraints. In *Automated Technology for Verification and Analysis - 17th International Symposium, ATVA 2019, Taipei, Taiwan, October 28-31, 2019, Proceedings (Lecture Notes in Computer Science)*, Yu-Fang Chen, Chih-Hong Cheng, and Javier Esparza (Eds.), Vol. 11781. Springer, 277–293.

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

[7] Pablo Barceló, Diego Figueira, and Leonid Libkin. 2013. Graph Logics with Rational Relations. *Logical Methods in Computer Science* 9, 3 (2013).

[8] Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings (Lecture Notes in Computer Science)*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.), Vol. 6806. Springer, 171–177.

[9] Murphy Berzish, Vijay Ganesh, and Yunhui Zheng. 2017. Z3str3: A string solver with theory-aware heuristics. In *2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017*, Daryl Stewart and Georg Weissenbacher (Eds.). IEEE, 55–59.

[10] Dmitry Blotsky, Federico Mora, Murphy Berzish, Yunhui Zheng, Ifaz Kabir, and Vijay Ganesh. 2018. StringFuzz: A Fuzzer for String Solvers. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II (Lecture Notes in Computer Science)*, Hana Chockler and Georg Weissenbacher (Eds.), Vol. 10982. Springer, 45–51.

[11] J. Richard Büchi and Steven Senger. 1988. Definability in the Existential Theory of Concatenation and Undecidable Extensions of this Theory. *Math. Log. Q.* 34, 4 (1988), 337–342.

[12] Taolue Chen, Yan Chen, Matthew Hague, Anthony W. Lin, and Zhilin Wu. 2018. What is decidable about string constraints with the ReplaceAll function. *PACMPL* 2, POPL (2018), 3:1–3:29.

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

[14] Joel D. Day, Vijay Ganesh, Paul He, Florin Manea, and Dirk Nowotka. 2018. The Satisfiability of Word Equations: Decidable and Undecidable Theories. In *Reachability Problems - 12th International Conference, RP 2018, Marseille, France, September 24-26, 2018, Proceedings (Lecture Notes*

in Computer Science), Igor Potapov and Pierre-Alain Reynier (Eds.), Vol. 11123. Springer, 15–29.

[15] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings (Lecture Notes in Computer Science)*, C. R. Ramakrishnan and Jakob Rehof (Eds.), Vol. 4963. Springer, 337–340.

[16] ECMA ECMAScript, European Computer Manufacturers Association, et al. 2019. Ecmascript language specification. https://www.ecma-international.org/ecma-262/

[17] Ting Gan, Mingshuai Chen, Liyun Dai, Bican Xia, and Naijun Zhan. 2015. Decidability of the Reachability for a Family of Linear Vector Fields. In *Automated Technology for Verification and Analysis - 13th International Symposium, ATVA 2015, Shanghai, China, October 12-15, 2015, Proceedings (Lecture Notes in Computer Science)*, Bernd Finkbeiner, Geguang Pu, and Lijun Zhang (Eds.), Vol. 9364. Springer, 482–499.

[18] Ting Gan, Mingshuai Chen, Yangjia Li, Bican Xia, and Naijun Zhan. 2018. Reachability Analysis for Solvable Dynamical Systems. *IEEE Trans. Automat. Contr.* 63, 7 (2018), 2003–2018.

[19] Vijay Ganesh and Murphy Berzish. 2016. Undecidability of a Theory of Strings, Linear Arithmetic over Length, and String-Number Conversion. *CoRR* abs/1605.09442 (2016). arXiv:1605.09442 http://arxiv.org/abs/1605.09442

[20] Vijay Ganesh, Mia Minnes, Armando Solar-Lezama, and Martin C. Rinard. 2012. Word Equations with Length Constraints: What's Decidable?. In *Hardware and Software: Verification and Testing - 8th International Haifa Verification Conference, HVC 2012, Haifa, Israel, November 6-8, 2012. Revised Selected Papers (Lecture Notes in Computer Science)*, Armin Biere, Amir Nahir, and Tanja E. J. Vos (Eds.), Vol. 7857. Springer, 209–226.

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

[22] Scott Kausler and Elena Sherman. 2014. Evaluation of string constraint solvers in the context of symbolic execution. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, Ivica Crnkovic, Marsha Chechik, and Paul Grünbacher (Eds.). ACM, 259–270.

[23] Adam Kiezun, Vijay Ganesh, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. 2009. HAMPI: a solver for string constraints. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ISSTA 2009, Chicago, IL, USA, July 19-23, 2009*, Gregg Rothermel and Laura K. Dillon (Eds.). ACM, 105–116.

[24] Zachary Kincaid, Jason Breck, John Cyphert, and Thomas W. Reps. 2019. Closed forms for numerical loops. *PACMPL* 3, POPL (2019), 55:1–55:29.

[25] Leetcode. 2019. LeetCode. https://leetcode.com/

[26] Guodong Li and Indradeep Ghosh. 2013. PASS: String Solving with Parameterized Array and Interval Automaton. In *Hardware and Software: Verification and Testing - 9th International Haifa Verification Conference, HVC 2013, Haifa, Israel, November 5-7, 2013, Proceedings (Lecture Notes in Computer Science)*, Valeria Bertacco and Axel Legay (Eds.), Vol. 8244. Springer, 15–31.

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

[28] Gennadiy Semenovich Makanin. 1977. The problem of solvability of equations in a free semigroup. *Matematicheskii Sbornik* 145, 2 (1977), 147–236.

[29] Yuri Matiyasevich. 2008. Computation Paradigms in Light of Hilbert's Tenth Problem. In *New Computational Paradigms: Changing Conceptions of What is Computable*, S. Barry Cooper, Benedikt Löwe, and Andrea Sorbi (Eds.). Springer New York, New York, NY, 59–85.

[30] Wojciech Plandowski. 1999. Satisfiability of Word Equations with Constants is in PSPACE. In *40th Annual Symposium on Foundations of Computer Science, FOCS '99, 17-18 October, 1999, New York, NY, USA*. IEEE Computer Society, 495–500.

[31] Wojciech Plandowski. 2006. An efficient algorithm for solving word equations. In *Proceedings of the 38th Annual ACM Symposium on Theory of Computing, Seattle, WA, USA, May 21-23, 2006*, Jon M. Kleinberg (Ed.). ACM, 467–476.

[32] Willard Van Orman Quine. 1946. Concatenation as a Basis for Arithmetic. *J. Symb. Log.* 11, 4 (1946), 105–114.

[33] Andrew Reynolds, Andres Nötzli, Clark W. Barrett, and Cesare Tinelli. 2019. High-Level Abstractions for Simplifying Extended String Constraints in SMT. In *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II (Lecture Notes in Computer Science)*, Isil Dillig and Serdar Tasiran (Eds.), Vol. 11562. Springer, 23–42.

[34] Andrew Reynolds, Maverick Woo, Clark W. Barrett, David Brumley, Tianyi Liang, and Cesare Tinelli. 2017. Scaling Up DPLL(T) String Solvers Using Context-Dependent Simplification. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II (Lecture Notes in Computer Science)*, Rupak Majumdar and Viktor Kuncak (Eds.), Vol. 10427. Springer, 453–474.

[35] John Michael Robson and Volker Diekert. 1999. On Quadratic Word Equations. In *STACS 99, 16th Annual Symposium on Theoretical Aspects of Computer Science, Trier, Germany, March 4-6, 1999, Proceedings (Lecture Notes in Computer Science)*, Christoph Meinel and Sophie Tison (Eds.), Vol. 1563. Springer, 217–226.

[36] José Fragoso Santos, Petar Maksimovic, Théotime Grohens, Julian Dolby, and Philippa Gardner. 2018. Symbolic Execution for JavaScript. In *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming, PPDP 2018, Frankfurt am Main, Germany, September 03-05, 2018*, David Sabel and Peter Thiemann (Eds.). ACM, 11:1–11:14.

[37] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. 2010. A Symbolic Execution Framework for JavaScript. In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*. IEEE Computer Society, 513–528.

[38] Prateek Saxena, Steve Hanna, Pongsin Poosankam, and Dawn Song. 2010. FLAX: Systematic Discovery of Client-side Validation Vulnerabilities in Rich Web Applications. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2010, San Diego, California, USA, 28th February - 3rd March 2010*. The Internet Society.

[39] Klaus U. Schulz. 1990. Makanin's Algorithm for Word Equations - Two Improvements and a Generalization. In *Word Equations and Related Topics, First International Workshop, IWWERT '90, Tübingen, Germany, October 1-3, 1990, Proceedings (Lecture Notes in Computer Science)*, Klaus U. Schulz (Ed.), Vol. 572. Springer, 85–150.

[40] Helmut Seidl, Thomas Schwentick, and Anca Muscholl. 2008. Counting in trees. In *Logic and Automata: History and Perspectives [in Honor of Wolfgang Thomas] (Texts in Logic and Games)*, Jörg Flum, Erich Grädel, and Thomas Wilke (Eds.), Vol. 2. Amsterdam University Press, 575–612.

[41] Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. [n.d.]. S3: A Symbolic String Solver for Vulnerability Detection in Web Applications. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, Gail-Joon Ahn, Moti Yung, and Ninghui Li (Eds.). ACM, 1232–1243.

[42] 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 (Lecture Notes in Computer Science)*, Swarat Chaudhuri and Azadeh Farzan (Eds.), Vol. 9779. Springer, 218–240.

[43] 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)*, Swarat Chaudhuri and Azadeh Farzan (Eds.), Vol. 9779. Springer, 241–260.

[44] Fang Yu, Muath Alkhalaf, and Tevfik Bultan. 2010. Stranger: An Automata-Based String Analysis Tool for PHP. In *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings (Lecture Notes in Computer Science)*, Javier Esparza and Rupak Majumdar (Eds.), Vol. 6015. Springer, 154–157.

[45] Fang Yu, Ching-Yuan Shueh, Chun-Han Lin, Yu-Fang Chen, Bow-Yaw Wang, and Tevfik Bultan. 2016. Optimal sanitization synthesis for web application vulnerability repair. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, Andreas Zeller and Abhik Roychoudhury (Eds.). ACM, 189–200.

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

[47] Yunhui Zheng, Xiangyu Zhang, and Vijay Ganesh. 2013. Z3-str: a z3-based string solver for web application analysis. 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*, Bertrand Meyer, Luciano Baresi, and Mira Mezini (Eds.). ACM, 114–124.

Solving String Constraints with Approximate Parikh Image

Petr Janků( ) and Lenka Turoňová

Faculty of Information Technology, Brno University of Technology, Brno, Czech Republic
{ijanku,ituronova}@fit.vutbr.cz

Abstract. In this paper, we propose a refined version of the Parikh image abstraction of finite automata to resolve string length constraints. We integrate this abstraction into the string solver SLOTH, where on top of handling length constraints, our abstraction is also used to speed-up solving other types of constraints. The experimental results show that our extension of SLOTH has good results on simple benchmarks as well as on complex benchmarks that are real-word combinations of transducer and concatenation constraints.

1 Introduction

Strings are a fundamental data type in many programming languages, 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 could easily lead to unexpected programming errors, e.g., cross-site scripting (a.k.a. XSS), which are ranked among the top three classes of web application security vulnerabilities by OWASP [11]. Some renowned companies like Google, Facebook, Adobe and Mozilla pay to whoever (hackers) finds a web application vulnerability such as cross-site scripting and SQL injection in their web applications [1], e.g., Google pays up to $10,000.

In recent years, there have been significant efforts on developing solvers for string constraints. Many rule-based solvers (such as Z3str2 [15], CVC4 [8], S3P [12]) are quite fast for the class of simple examples that they can handle. They are sound but do not guarantee termination. Other tools for dealing with string constraints (such as Norn [1], Sloth [6], Ostrich [4]) are based on automata. They use decision procedures which work with fragments of logic over string constraints that are rich enough to be usable in real-world web applications.

This work has been supported by the Czech Science Foundation (project No. 17-12465S), the IT4Innovations Excellence in Science (project No. LQ1602), and the FIT BUT internal projects FIT-S-17-4014 and FEKT/FIT-J-19-5906.
We thank you to Lukáš Holík for all the support and encouragement he gave us and also the time he spent with us during discussions.

[1] For more information, see https://www.netsparker.com/blog/web-security/google-increase-reward-vulnerability-program-xss/.

© Springer Nature Switzerland AG 2020
R. Moreno-Díaz et al. (Eds.): EUROCAST 2019, LNCS 12013, pp. 491–498, 2020.
https://doi.org/10.1007/978-3-030-45093-9_59

They are sound and complete. SLOTH was the first solver that can handle string constraints including transducers, however, unlike NORN and OSTRICH it is not able to handle length constraints yet. Moreover, these tools are not efficient on simple benchmarks as the rule-based solvers above.

Example 1. The following JavaScript snippet is an adaptation of an example from [2,7]:

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

This is a typical example of string manipulation in a web application. The code attempts to first sanitise the value of name using the sanitization functions htmlEscape and escapeString from the Closure Library [5]. The author of this code accidentally swapped the order of the two first lines. Due to this subtle mistake, the code is vulnerable to XSS, because the variable y may be assigned an unsafe value. To detect such mistakes, we have to first translate the program and the safety property to a string constraint, which is satisfiable if and only if y can be assigned an unsafe value. However, if we would add the length constraints (e.g. `x.length == 2*y.lenght;`) to the code, none of SLOTH, OSTRICH, or NORN would be able to handle them.

The length constraints are quite common in programs like this. Hence, in this paper, we present how to extend the method of SLOTH to be able to cope with them. Our decision procedure is based on the computation of Parikh images for automata representing constraint functions. Parikh image maps each symbol to the number of occurrences in the string regardless to its position.

For one nonderministic finite automaton, one can easily computate the Parikh image by standard automata procedures. However, to compute an exact Parikh image for a whole formula of contraints is demanding. The existing solution proposes first to compute the product of the automata representing the subformulea and then compute the Parikh image of their product. Unfortunately, the exact computation of the Parikh images is computationally far too expensive. Even more importantly, the resulting semilinear expressions become exponential to the number of automata.

We therefore propose a decision procedure which computes an over-approximation of the exact solution that is sufficiently close to the exact solution. We first compute the membership Parikh images of the automata representing the string constraints. Then we use concatenation and substitution to compute the over-approximation of the Parikh image of the whole formula. However, we will not get the same result as with the previous approach since the Parikh image forgets the ordering of the symbols in the world. This causes that we could accept even words that are not accepted by the first approach. But even though our method does not provide accurate results, it is able to handle the lenght constraints and solve also real-world cases.

Outline. Our paper is organized as follow. In Sect. 1, we introduce relevant notions from logic and automata theory. Section 3 presents an introduction to a string language. Section 4 explains the notion of Parikh image and operations on Parikh images. Section 5 presents the main decision procedure. In Sect. 6, the experimental results are presented.

2 Preliminaries

Bit Vector. Let $\mathbb{B} = \{0,1\}$ be a set of Boolean values and V a finite set of bit variables. *Bit vectors* are defined as functions $b : V \rightarrow \mathbb{B}$. In this paper, bit vectors are described by conjunctions of literals over V. We will denote the set of all bit vectors over V by $\mathbb{P}(V)$ and a set of all formulae over V by $\mathbb{F}_V$.

Further, let $k \geq 1$, and let $V\langle k\rangle = V \times [k]$ where $[k]$ denotes the set $\{1,\ldots,k\}$. Given a word $w = b_1^k \ldots b_m^k \in \mathbb{P}(V\langle k\rangle)^*$ over bit vectors, we denote by $b_j^k[i] \times \{i\} = b_j^k \cap (V \times \{i\})$, $1 \leq j \leq m$ where $b_j^k[i] \in \mathbb{P}(V)$ the j-th bit vector of the i-th track. Further, $w[i] \in \mathbb{P}(V)^*$ such that $w[i] = b_1^k[i] \ldots b_m^k[i]$ is the word which keeps the content of the i-th track of w only. For a bit vector $b \in \mathbb{P}(V)$, we denote by $\{b\}$ the set of variables in the vector.

Automata and Transducers. A *succinct nondeterministic finite automaton* (NFA) over bit variables V is a tuple $\mathcal{A} = (V, Q, \Delta, q_0, F)$ where Q is a finite set of *states*, $\Delta \subseteq Q \times \mathbb{F}_V \times Q$ is *transition relation*, $q_0 \in Q$ is an *initial state*, and $F \subseteq Q$ is a finite set of *final states*. $\mathcal{A}$ accepts a word w iff there is a sequence $q_0 b_1^k q_1 \ldots b_m^k q_m$ where $b_i^k \in \mathbb{P}(V)$ for every $1 \leq i \leq m$ such that $(q_i, \varphi_i, q_{i-1}) \in \Delta$ for every $1 \leq i \leq m$ where $b_i^k \models \varphi_i$, $q_m \in F$, and $w = b_1^k \ \ldots b_m^k \in \mathbb{P}(V)^*$, $m \geq 0$, where each b_i^k, $1 \leq i \leq m$, is a bit vector encoding the i-th letter of w. The *language* of $\mathcal{A}$ is the set $L(\mathcal{A})$ of accepted words.

A k-track *succinct finite automaton* over V is an automaton $\mathcal{R}\langle k\rangle = (V\langle k\rangle, Q, \Delta, I, F)$, $k \geq 1$. The relation $R(\mathcal{R}\langle k\rangle) \subseteq (\mathbb{P}(V)^*)^k$ *recognised* by $\mathcal{R}$ contains a k-tuple of words $(x_1,\ldots,x_k)$ over $\mathbb{P}(V)$ iff there is a word $w \in L(\mathcal{R})$ such that $x_i = w[i]$ for each $1 \leq i \leq k$. A *finite transducer* (FT) $\mathcal{R}$ is a 2-track automaton.

Strings and Languages. We assume a finite alphabet Σ. Σ^* represents a set of finite words over Σ, where the empty word is denoted by ϵ. Let x and y be finite words in Σ^*. The concatenation of x and y is denoted by $x \circ y$. We denote by $|x|$ the length of a word $x \in \Sigma^*$. A language is a subset of Σ^*. The concatenation of languages L, L' is the language $L \circ L' = x \circ x' | x \in L \wedge x' \in L'$, and the iteration L^* of L is the smallest language closed under $\circ$ and containing L and ϵ.

3 String Language

Let $\mathbb{X}$ be a set of variables and x, y be *string variables* ranging over Σ^*. A *string formula* over string terms $\{t_{str}\}^*$ is a Boolean combination of *word equations* $x =$

t_{str} whose right-hand side t_{str} might contain the concatenation operator, *regular constraints* P, *rational constraints* R and arithmetic inequalities:

$$\begin{aligned} \varphi &::= x = t_{str} \mid P(x) \mid R(x,y) \mid t_{ar} \geq t_{ar} \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \neg\varphi \\ t_{str} &::= x \mid \epsilon \mid t_{str} \circ t_{str} \\ t_{ar} &::= k \mid |t_{str}| \mid t_{ar} + t_{ar} \end{aligned}$$

In the grammar, x ranges over string variables, $R \subseteq (\Sigma^*)^2$ is assumed to be a binary rational relation on words of Σ^*, and $P \subseteq \Sigma^*$ is a regular language. We will represent regular languages by succinct automata and tranducers denoted as $\mathcal{R}$ and $\mathcal{A}$, respectively. The arithmetic terms t_{ar} are linear functions over term lengths and integers, and arithmetic constraints are inequalities of arithmetic terms. The set of word variables appearing in a term is defined as follows: $\mathit{Vars}(\epsilon) = \emptyset$, $\mathit{Vars}(c) = \emptyset$, $\mathit{Vars}(u) = \{u\}$ and $\mathit{Vars}(t_1 \circ t_2) = \mathit{Vars}(t_1) \cup \mathit{Vars}(t_2)$.

To simplify the representation, we do not consider *mixed* string terms t_{str} that contain, besides variables of $\mathbb{X}$, also symbols of Σ. This is without loss of generality since a mixed term can be encoded as a conjunction of the pure terms over $\mathbb{X}$ obtained by replacing every occurrence of a letter $a \in \Sigma$ by a fresh variable x, and adding a regular membership constraint $\mathcal{A}_a(x)$ with $L(\mathcal{A}_a) = \{a\}$.

Semantics. A formula φ is interpreted over an *assignment* $\iota : \mathbb{X}_\varphi \to \Sigma^*$ of its variables $\mathbb{X}_\varphi$ to strings over Σ^*. ι is extended to string terms by $\iota(t_{s_1} \circ t_{s_2}) = \iota(t_{s_1}) \circ \iota(t_{s_2})$ and to arithmetic terms by $\iota(|t_s|) = |\iota(t_s)|$, $\iota(k) = k$ and $\iota(t_i + t_i') = \iota(t_i) + \iota(t_i')$. We formalize the satisfaction relation for word equations, regular constraints, rational constraints, and arithmetic inequalities, assuming the standard meaning of Boolean connectives:

$$\begin{aligned} x = t_{str} &\text{ iff } \iota(x) = \iota(t_{str}) \\ \iota(P(x)) = \top &\text{ iff } \iota(x) \in P \\ \iota(\mathcal{R}(x,y)) = \top &\text{ iff } (\iota(x), \iota(y)) \in \mathcal{R} \\ \iota(t_{i_1} \leq t_{i_2}) = \top &\text{ iff } \iota(t_{i_1}) \leq \iota(t_{i_2}) \end{aligned}$$

The truth value of Boolean combinations of formulae under ι is defined as usual. If $\iota(\varphi) = \top$ then ι is a *solution* of φ, written $\iota \models \varphi$. The formula φ is *satisfiable* iff it has a solution, otherwise it is *unsatisfiable*.

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 a rational transducer $\mathcal{R}$ [10]. Therefore, we restrict the formulae to be in so-called *straight-line form*. The definition of *straight-line fragment* as well as a linear-time algorithm for checking whether a formula φ falls into the straight-line fragment is defined in [9].

4 Parikh Image

The Parikh image of a string abstracts from the ordering in the string. Particularly, the Parikh image of a string x maps each symbol a to the number of its

occurrences in the string x (regardless to their position). Parikh image of a given language is then the set of Parikh images of the words of the language.

In this chapter, we present a construction of the Parikh image of a given NFA $\mathcal{A} = (V, Q, \Delta, q_0, F)$. The algorithm is modified version of the algorithm from [13] which computes the Parikh image for a given context-free grammar G. This algorithm contains a small mistake that has been fixed by Barner in a 2006 Master's thesis [3]. Since for every regular grammar there exists a corresponding NFA, we can easily customize the algorithm for NFA such that one can compute an existential Presburger formula $\phi_{\mathcal{A}}$ which characterizes the Parikh image of the language $L(\mathcal{A})$ recognized by $\mathcal{A}$ in the following way.

Let us define a variable $\#_\varphi$ for each $\varphi \in \mathbb{F}_V$, y_t for each $t \in \Delta$, and u_q for each $q \in Q$, respectively. The free variables of $\phi_{\mathcal{A}}$ are variables $\#_\varphi$ and we write *Free*$(\phi_{\mathcal{A}})$ for the set of all free variables in the formula $\phi_{\mathcal{A}}$. The formula $\phi_{\mathcal{A}}$ is the conjunction of the following three kinds of formulae:

- $u_q + \sum_{t=(q',\varphi,q)\in\Delta} y_t - \sum_{t=(q,\varphi,q')\in\Delta} y_t = 0$ for each $q \in Q$, where the variable u_q is restricted as follows: $u_{q_0} = 1$, $u_{q_F} \in \{0, -1\}$ for $q_F \in F$, and $u_q = 0$ for all other $q \in Q \setminus (\{q_0\} \cup F)$.
- $y_t \geq 0$ for each $t \in \Delta$ since the variable y_t cannot be assigned a negative value.
- $\#_\varphi = \sum_{t=(q,\varphi,q')\in\Delta} y_t$ for each $\varphi \in \mathbb{F}$ to ensure that the value x_φ are consistent with the y_t.
- To express the connectedness of the automaton, we use an additional variable z_q for each $q \in Q$ which reflects the distance of q from q_0 in a spanning tree on the subgraph of $\mathcal{A}$ induced by those $t \in \Delta$ with $y_t \geq 0$. To this end, we add for each $q \in Q$ a formulae $z_q = 1 \wedge y_t \geq 0$ if q is an initial state, otherwise $(z_q = 0 \wedge \bigwedge_{t\in\Delta_q^+} y_t = 0) \vee \bigvee_{t\in\Delta_q^+}(y_t \geq 0 \wedge z_{q'} \geq 0 \wedge z_q = z_{q'} + 1)$ where $\Delta_q^+ = \{(q', \varphi, q) \in \Delta\}$ is a set of ingoing transitions.

The resulting existential Presburger formula is then $\exists z_{q_1}, \ldots, z_{q_n}, u_{q_1}, \ldots, u_{q_n}, y_{t_1}, \ldots, z_{t_m} : \phi_{\mathcal{A}}$ where n is the number of states and m is the number of transitions of the given automaton. This algorithm can be directly applied to transducers where the free variables are $\#_\varphi$ such that $\varphi \in \mathbb{F}_{V\langle 2\rangle}$.

4.1 Operations on Parikh Images

In our decision procedure, we will need to use *projection* of the Parikh image of transducers and *intersection* of Parikh images. We have to find a way how to deal with alphabet predicates of transducers since our version of the intersection of Parikh images works only with alphabet predicates over a non-indexed set of bit variables. The intersection of Parikh images is needed since the alphabet predicates of one automaton can represent a set of symbols which may contains common symbols for more than one automaton. These operations can be implemented in linear space and time.

Projection. Let $\mathcal{R} = (V\langle 2\rangle, Q, \Delta, I, F)$ be a transducer representing a constraint $R(x, y)$ and let $\varphi \in \mathbb{F}_{V\langle 2\rangle}$ be a formula over $\{b^k\} \in 2^{V\langle 2\rangle}$ where $b^k \in \mathbb{P}(V\langle 2\rangle)$. We write $\varphi[x]$ to denote a alphabet projection of φ where $\varphi[x]$ is the subformula of φ such that only contains bits from $b^k[i]$ and i is the position of x in R. Given the Parikh image $\phi_{\mathcal{R}}$ of $\mathcal{R}$, we denote by $\phi_{\mathcal{R}}[x]$ a *projection* of $\phi_{\mathcal{R}}$ where the set of free variables is $\mathit{Free}(\phi_{\mathcal{R}}[x]) = \{\#_{\varphi[x]} \mid \#_\varphi \in \mathit{Free}(\phi_{\mathcal{R}})\}$. Further, we need to introduce the auxiliary function λ that assigns to each variable $\#_{\varphi[x]}$ a set $\{\#_{\varphi'} \mid \varphi'[x] = \varphi[x]\}$. The resulting formula of projection $\phi_{\mathcal{R}}$ has then the form $\phi_{\mathcal{R}}[x] = \exists \#_{\varphi_1}, \ldots, \#_{\varphi_n} : \phi_{\mathcal{R}} \wedge \bigwedge_{\#_{\varphi[x]} \in \mathit{Free}(\phi_{\mathcal{R}}[x])} \left(\#_{\varphi[x]} = \sum_{\#_\varphi \in \lambda(\#_{\varphi[x]})} \#_\varphi\right)$ where $\#_{\varphi_i} \in \mathit{Free}(\phi_{\mathcal{R}})$ for $1 \leq i \leq n$.

Intersection. We assume that both Parikh images have alphabet predicates $\mathbb{F}_V$ over the same set of bit variables V. Given two Parikh images ϕ_1 and ϕ_2, their intersection $\phi_\curlywedge = \phi_1 \curlywedge \phi_2$ can be constructed as follows. First, we compute a set of fresh variables $\mathcal{I} = \{\#_{\varphi_1 \curlywedge \varphi_2} \mid \#_{\varphi_1} \in \mathit{Free}(\phi_1) \wedge \#_{\varphi_2} \in \mathit{Free}(\phi_2) \wedge \exists b \in \mathbb{P}(V) : b \models \varphi_1 \wedge \varphi_2\}$ representing the number of common symbols for ϕ_1 and ϕ_2. Next, we define for each Parikh image ϕ_i a function $\tau_i : \mathit{Free}(\phi_i) \to 2^{\mathcal{I}}$ such that $\tau_1(\#_{\varphi_1}) = \{\#_{\varphi_1 \curlywedge \varphi_2} \in \mathcal{I}\}$ and $\tau_2(\#_{\varphi_2}) = \{\#_{\varphi_1 \curlywedge \varphi_2} \in \mathcal{I}\}$. Finally, the intersection is define as $\phi_\curlywedge = \phi_1 \wedge \phi_2 \wedge \bigwedge_{\#_{\varphi_1} \in \mathit{Free}(\phi_1)} \left(\#_{\varphi_1} = \sum_{\#_{\varphi_1'} \in \tau_1(\#_{\varphi_1})} \#_{\varphi_1'}\right) \wedge \bigwedge_{\#_{\varphi_2} \in \mathit{Free}(\phi_2)} \left(\#_{\varphi_2} = \sum_{\#_{\varphi_2'} \in \tau_2(\#_{\varphi_2})} \#_{\varphi_2'}\right)$.

5 Decision Procedure

Our decision procedure is based on computation of the Parikh images of the automata representing string constraints. Let $\varphi := \varphi_{cstr} \wedge \varphi_{eq} \wedge \varphi_{ar}$ be a formula in straight-line form where φ_{cstr} is a conjunction of regular constraints (or their negation) and rational constraints, φ_{eq} is a conjunction of word equations of the form $x = y_1 \circ y_2 \circ \cdots \circ y_n$, and φ_{ar} is a conjunction of arithmetic inequalities. The result of the decision procedure is an existential Presburger formula ϕ_φ which represents an over-approximation of the Parikh image of φ.

We assume that each variable $x \in \mathit{Vars}(\varphi)$ is restricted by an automaton or a transducer. Note that the function $\mathit{Vars}(\varphi)$ denotes a set of variables appearing in the formula φ. We write $\mathbb{T}$ to denote a set of Parikh images. The procedure is divided into three steps as follows.

- **Step 1:** First, we compute Parikh images of automata and transducers representing the constraints from φ_{cstr} using the algorithm from Sect. 4. We define a mapping $\rho_{cstr} : \mathit{Vars}(\varphi_{cstr}) \to \mathbb{T}$ that maps each string variable $x \in \mathit{Vars}(\varphi_{cstr})$ to the over-approximation of its Parikh image. Let $P_1(x), \ldots, P_n(x)$ and $R_1(x, y), \ldots, R_m(x, y)$ be constraints from φ_{cstr} restricting x. A formula ϕ_x representing the Parikh image of x is then computed using the algorithm from Sect. 4.1 as $\phi_x = \phi_{\mathcal{A}_x} \curlywedge \phi_{\mathcal{A}_1} \curlywedge \ldots \curlywedge \phi_{\mathcal{A}_n} \curlywedge \phi_{\mathcal{R}_1}[x] \curlywedge \ldots \curlywedge \phi_{\mathcal{R}_m}[x]$ where $\phi_{\mathcal{A}_i}$, $0 \leq i \leq n$, is the Parikh image of the automaton $\mathcal{A}_i$ representing $P_i(x)$ and $\phi_{\mathcal{R}_j}$, $0 \leq j \leq m$, is the Parikh image of the transducer $\mathcal{R}_j$ representing $R_j(x, y)$.

– **Step 2:** We define a mapping $\rho_{eq} : Vars(\varphi_{eq}) \to \mathbb{T}$ that maps each string variable $x \in Vars(\varphi_{eq})$ to the over-approximation of its Parikh image as $\phi_x = (\bigwedge_{i=1}^{k} \rho_{cstr}(y_i) \wedge \bigwedge_{j=k+1}^{n} \rho_{eq}(y_j)) \curlywedge \rho_{cstr}(x)$. We assume that $Free(y_1) \cap \cdots \cap Free(y_n) = \emptyset$. This can be done by adding double negation to the alphabet predicates which helps to distinguish free variables of individual y_i. Parikh image does not preserve the ordering of the symbols in the string, therefore, we can reorder the right side of the equation $y_1 \circ \cdots \circ y_k \circ \cdots \circ y_n$ such that $\forall 1 \leq i \leq k : y_i \in \varphi_{cstr}$ and $\forall k \leq j \leq n : y_j \in \varphi_{eq}$. Moreover, the reordering can be done in such a way that each variable on the right side of the equation is already defined since φ falls into the straight-line fragment.
– **Step 3:** Finally, we build the final formula ϕ_φ using mappings ρ_{cstr} and ρ_{eq}. Let $\mathbb{X}_{eq} \subseteq Vars(\varphi_{eq})$ be a set of all variables that are on the left side of the equations. The resulting formula ϕ_φ is then a conjunction $\phi_\varphi = (\curlywedge_{x \in \mathbb{X}_{eq}} \rho_{eq}(x)) \curlywedge (\curlywedge_{x \in Vars(\varphi) \setminus \mathbb{X}_{eq}} \rho_{cstr}(x))$.

6 Experiments

We have implemented our decision procedure extending the method of Sloth [6] as a tool, called PICoSo. Sloth is a decision procedure for the straight-line fragment and acyclic formulas. It uses succinct alternating finite-state automata as concise symbolic representation of string constraints. Like Sloth, PICoSo was implemented in Scala.

To evaluate its performance, we compared PICoSo against Sloth. We performed experiments on benchmarks with diverse characteristics.

Table 1. Performance of PICoSo in comparison to Sloth.

		Sloth	PICoSo
Norn (1027)	sat (sec)	**314 (545)**	313 (566)
	unsat (sec)	353 (624)	**356 (602)**
	timeout	0	0
	error/un	360	**358**
SLOG (3392)	sat (sec)	922 (**5526**)	**923** (5801)
	unsat (sec)	2033 (5950)	**2080 (4382)**
	timeout	437	**389**
	error/un	0	0
SLOG-LEN (394)	sat (sec)	0	0
	unsat (sec)	266 (659)	**296 (773)**
	timeout	**4**	15
	error/un	124	**83**

The first set of benchmarks is obtained from Norn group [1] and implements string manipulating functions such as the Hamming and Levenshtein distances. It consists of small test case that is combinations of concatenations, regular constraints, and length constraints. The second set SLOG [14] is derived from the security analysis of real web applications. It contains regular constraints, concatenations, and transducer constraints such as `Replace` but no length constraints. The last set is obtained from SLOG by selecting 394 examples containing `Replace` operation. It was extended by `RelaceAll` operation and since in practice, it is common to restrict the size of string variables in web applications, we added length constraints of the form $|x| + |y|Rn$, where $R \in \{=, <, >\}$, $n \in \{4, 8, 12, 16, 20\}$, and x, y are string variables.

The summary of the experiments is shown in Table 1. All experiments were executed on a computer with Intel Xeon E5-2630v2 CPU @ 2.60 GHz and 32 GiB

RAM. The time limit was 30 s was imposed on each test case. The rows indicate the number of times the solver returned satisfiable/unsatisfiable (sat/unsat), the number of times the solver ran out of 30-s limit (timeout), and the number of times the solver either crashed or returned unknown (error/un).

The results show that PICoSo outperforms Sloth on all of unsat examples. Sloth is however slightly better in case of sat examples due to the addition computation of the over-approximation of the Parikh image. Sloth timed out on 441 cases while PICoSo run out of time only in 404 cases. This shows that our proposed procedure is efficient in solving not only length constraints, but also other types of constraints.

References

1. Abdulla, P.A., et al.: String constraints for verification. In: CAV 2014, pp. 150–166 (2014)
2. Barceló, P., Figueira, D., Libkin, L.: Graph logics with rational relations. Proc. ACM Program. Lang. **9**, 30 (2013)
3. Barner, S.: H3 mit gleichheitstheorien. Master's thesis, Technical University of Munich, Germany (2006)
4. Chen, T., Hague, M., Lin, A.W., Rümmer, P., Wu, Z.: Decision procedures for path feasibility of string-manipulating programs with complex operations. Proc. ACM Program. Lang. **3**, 49:1–49:30 (2019)
5. G. co. 2015. Google closure library (referred in Nov 2015) (2015). https://developers.google.com/closure/library/
6. Holík, L., Janků, P., Lin, A.W., Rümmer, P., Vojnar, T.: String constraints with concatenation and transducers solved efficiently. PACMPL **2**(POPL), 1–32 (2018)
7. Kern, C.: Securing the tangled web. ACM **57**, 38–47 (2014)
8. Liang, T., Reynolds, A., Tinelli, C., Barrett, C., Deters, M.: A DPLL(T) theory solver for a theory of strings and regular expressions. In: CAV 2014 (2014)
9. Lin, A.W., Barceló, P.: String solving with word equations and transducers: towards a logic for analysing mutation XSS. In: POPL, pp. 123–136 (2016)
10. Morvan, C.: On rational graphs. In: Tiuryn, J. (ed.) FoSSaCS 2000. LNCS, vol. 1784, pp. 252–266. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-46432-8_17
11. OWASP: The ten most critical web application security risks (2013). https://www.owasp.org/images/f/f8/OWASP_Top_10_-_2013.pdf
12. Trinh, M., Chu, D., Jaffar, J.: Progressive reasoning over recursively-defined strings. In: CAV 2016, pp. 218–240 (2016)
13. Verma, K.N., Seidl, H., Schwentick, T.: On the complexity of equational horn clauses. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS (LNAI), vol. 3632, pp. 337–352. Springer, Heidelberg (2005). https://doi.org/10.1007/11532231_25
14. Wang, H.-E., Tsai, T.-L., Lin, C.-H., Yu, F., Jiang, J.-H.R.: String analysis via automata manipulation with logic circuit representation. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9779, pp. 241–260. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41528-4_13
15. Zheng, Y., et al.: Z3str2: an efficient solver for strings, regular expressions, and length constraints. Formal Meth. Syst. Des. **50**(2–3), 249–288 (2014)